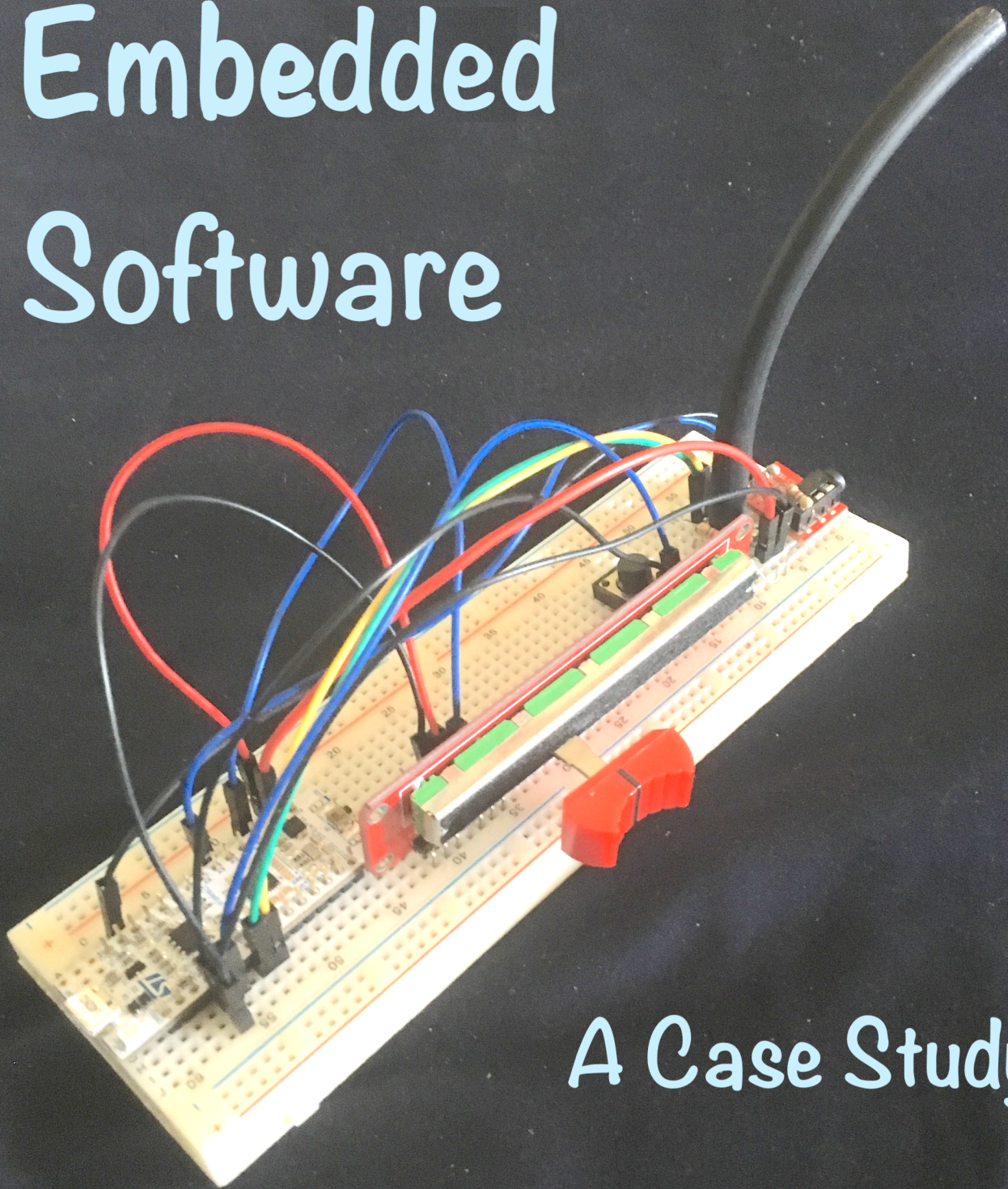


How To Develop Embedded Software



A Case Study

VISIT...

LANZAROTE
Caliente.COM

How To Develop Embedded Software

A Case Study

David Clifton

This book is for sale at <http://leanpub.com/howtodevelopembeddedsoftware>

This version was published on 2020-06-12



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2014 - 2020 David Clifton

Contents

Chapter 1: Preliminaries	1
Introduction	1
Some Definitions	1
Chapter 2: The Conceptual Map	3
Documentary Aspect of the Conceptual Map	3
Mental Aspect of the Conceptual Map	3
Layers of the Conceptual Map	4
Real Engineers	6
Chapter 3: Development Processes	8
Chapter 4: Context	11
Chapter 5: Requirements Analysis	13
Use Case Summary	13
Formal Requirements List	15
Reverse Engineering	18
Requirements Workshop	19
Requirements Model	20
Safety Issues	20
Chapter 6: Preliminary Design	22
Digital Trombone Architecture	22
Architecture Workshop	30
Chapter 7: Estimating	31
COCOMO Method	31
Modified Function Point Method	32
Add Up The Guesses Method	33
Combining The Estimates	34
Chapter 8: Tool Selection	35
Chapter 9: Hardware Support	37

CONTENTS

Chapter 10: Detailed Design	39
Runtime Pattern for Digital Trombone	39
Module Overview Diagram for Digital Trombone	40
Module Interaction Diagrams for Digital Trombone	41
Module Catalog	44
Chapter 11: Code	57
Choice of Language	57
Choice of Hardware Access Layer	58
Coding Standards	59
Coding the Trombone Project	60
Chapter 12: Debug	61
Bug Categories	61
Chapter 13: Integrate	66
Advice for Integrators	66
Chapter 14: Verify	68
Chapter 15: Validate	78
APPENDIX A	83
Cellular Calculator Scripts for Estimating	83
APPENDIX B	95
Vendor Documents	95
Bibliography	96

Chapter 1: Preliminaries

Introduction

This book is a case study of an embedded software project. It describes a project from start to finish, and all of the material and documents developed and used along the way.

The information supplied comes directly from the author's 30 years of experience developing embedded software for electronic products made by top U.S. corporations.

The example system is a audio synthesizer, similar to a Trombone. The original acoustic Trombone required the user to blow with vibrating lips into a mouthpiece, and slide a variable column of air back and forth. It responded by producing musical tones of varying pitch and volume.

The digital version has the user blow into a mouthpiece with varying amounts of pressure, and slide a linear potentiometer back and forth to produce musical tones of varying pitch and volume. This simple example is challenging enough to illustrate some common embedded development practices.

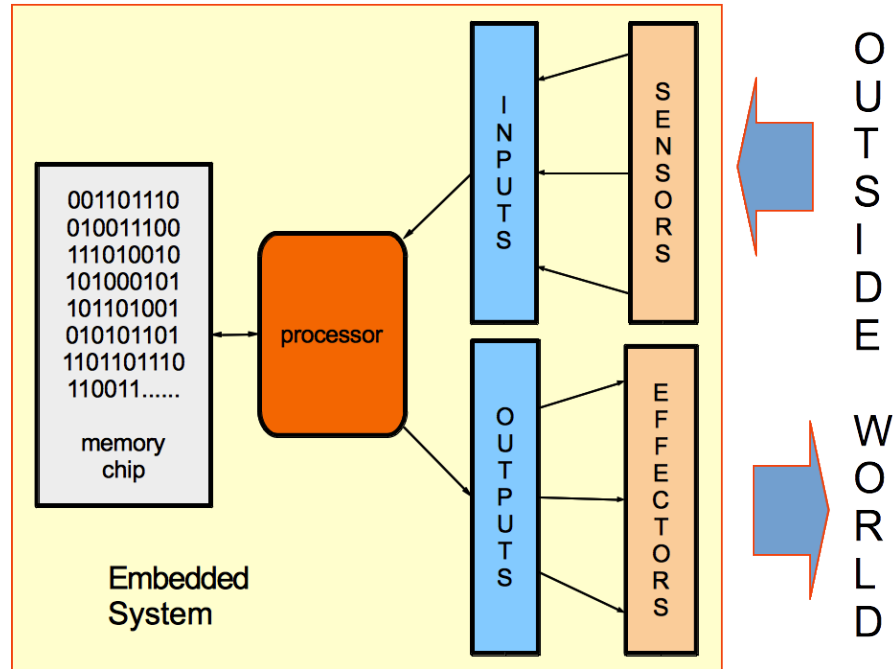
This book is written with three types of readers in mind. First is the experienced PC programmer who wishes to transition into the world of embedded systems.

Another is the electronic engineer who writes code but has not yet had much exposure to the software side of architecture and design.

Finally, the hobbyist, who works well within the Arduino or similar environment, can learn additional concepts that will help him take the next step into software engineering.

Some Definitions

An embedded system is an information appliance that uses sensors and effectors to interact with the external environment. A microprocessor monitors inputs from the sensors and produces outputs to the effectors. Software guides its production of outputs from the internal state and available inputs.



The memory, processor, input and output peripherals, sensors and effectors comprise the hardware of the embedded system. The bit sequence in the memory chip is called the software of the embedded system.

Embedded systems are made up of discrete hardware components and software components.

A hardware component is made out of material parts. Software components are compiled from source code modules.

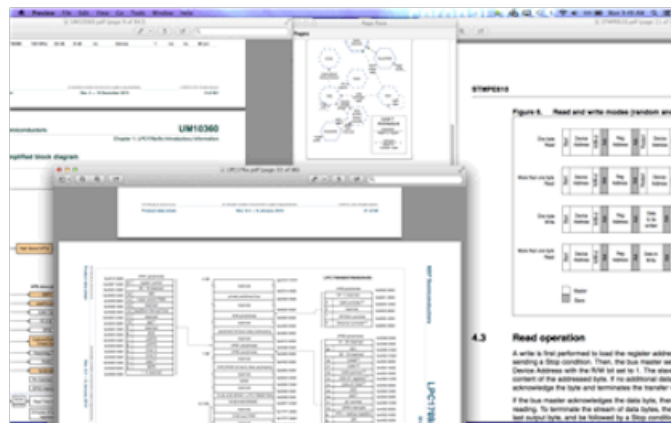
A source code module is a named collection of functions or methods which implements services for other modules, and may encapsulate local data structures. When compiled and loaded, each module can interact with hardware components and other software modules in a manner determined by its design, and consistent with the system requirements.

Chapter 2: The Conceptual Map

Software developers make bit sequences that go into memory chips. A typical developer cannot listen to a description of desired system behavior, and then just dash off a bit sequence that does the trick. Instead, he must create a conceptual map of the problem to be solved, and all of the elements needed to produce the necessary bit sequence. The conceptual map has both a documentary and a mental aspect.

Documentary Aspect of the Conceptual Map

The documentary aspect of the conceptual map is a collection of data sheets, websites, analysis and design documents, interviews, and meeting minutes which bear on the project. These documents are sometimes organized in a project folder that makes them readily accessible to each member of the team.



modern documentary map

For links to vendor documents which are part of the documentary map of the digital Trombone project, see Appendix B.

Mental Aspect of the Conceptual Map

The mental aspect of the conceptual map is a representation in each team member's mind of the personnel, environment, goals, hardware, and software objects involved in a development project, and the ways in which they interact.



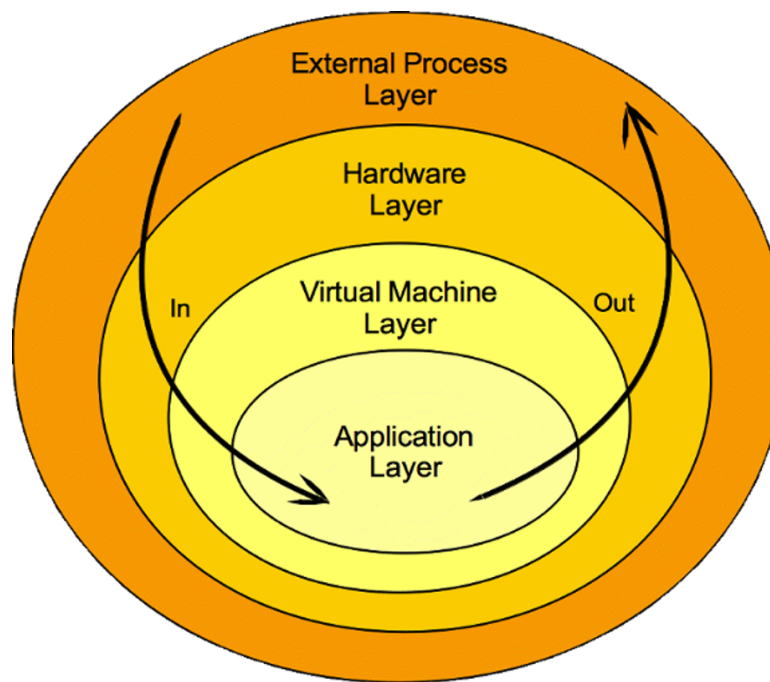
Pessimistic Impression of the mental conceptual map

The dreaded “learning curve” of an embedded project is the time necessary for a new team member to create, from conversations with team members, and from the project’s documentary map, a mental map for the project.

Layers of the Conceptual Map

We can subdivide the conceptual map into different layers. Each layer deals with different aspects of the mapping between the external world, and the embedded software, which interacts with it.

The diagram below shows such a subdivision of a project’s conceptual map:



In this diagram, the subdivisions of the conceptual map relate to the flow of information or influence in the embedded system.

Influence flows in from the external process layer through the hardware layer, where it is transformed into information, which flows into the virtual machine layer, and is processed by the application layer.

Response information is sent back out via the virtual machine layer, and is transformed by the hardware layer into influence on the external process layer.

In the external process layer, the embedded system is considered to be a unit which exchanges influences with the external world. The concepts in this layer describe, in a general way, each of those interactions. They also describe, in verifiable terms, every aspect of those interactions which motivated the building of the system.

The hardware layer of the conceptual map contains data sheets, schematics, and concepts related to the operation of the embedded system hardware. Documentation of the hardware is provided by hardware vendors and designers for the circuit board(s), processor, peripherals, sensors, effectors, memory chips, comm chips, and power supplies.

The layer between the hardware and the application is called the virtual machine layer. It describes the virtual (software-simulated) objects through which the application accesses the hardware, and organizes itself in time. These virtual objects include the primitives of the programming language, the operating system (if any), and any code libraries supplied with the compiler or operating system or written separately by team members.

The application layer is a collection of virtual objects which interact to model the external process and/or exchange influences with it via the virtual machine and hardware layers. This layer is mostly

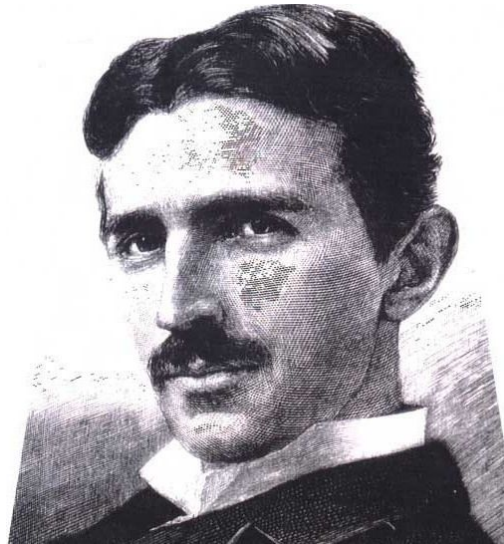
invented and documented by the developers. It may also include virtual objects supplied in code libraries by the system sponsor, or by third party vendors.

The quality of the conceptual map, in both its physical and mental forms, figures prominently in the success or failure of an embedded project.

If the team members have created a well organized collection of documents, and if they take time to read and understand them; the project has a better prospect of success.

Real Engineers

A belief persists among some developers, typically the more hardware oriented, that real engineers don't write documentation. There is some justification for this position. For a small system, one not too complex; it is possible to write code without extensively documenting the conceptual map.



Nikola Tesla – Photo Courtesy of Tesla Memorial Society of New York

Nikola Tesla, arguably the Earth's greatest engineer, is widely believed to have kept most of his plans and designs in his head. It is said he could set simulations running in his mind, and revisit them from time to time to check for wear and tear on components.

Mr. Tesla was, without a doubt, a master of the mental conceptual map. Even so, he kept extensive lab notes, and did in fact document his work in patent applications.

If the choice is made not to document, troubles may arise when any of the following conditions are met:

- The number of callable functions exceeds a hundred.
- There is frequent turnover in the engineering staff.
- Safety and regulatory issues require full disclosure of the development process.

Throughout this book, various methods will be used to create and document the conceptual map, be it in requirements, preliminary, or detailed design documents. If the reader is convinced that documenting gilds the lily, then he will probably ignore these methods.

Just as some engineers favor sparse documentation, others prefer rich documentation. The latter group are persons who require more rigor than that supplied in the methods used here. Those readers will no doubt feel free to use their own methods of documenting the conceptual map, or adopt methods that may be found in abundance in the literature.

Chapter 3: Development Processes

The software developer's task is to complete the conceptual map of the embedded system, and to implement and test its software modules on the target hardware. This leads him to discover and document the system's required interactions with the outer world, learn how the hardware works, understand the virtual machine, invent the interacting software modules of the application, debug, test, and validate the system. To these ends, the developer participates in the processes described below.

Establish Context

Become familiar with the physical and social context in which the system will be developed.

Requirements Analysis

Describe the external process and the proposed useful ways of interacting with it. Formulate a complete set of goals for the embedded system. Digest the documentation from hardware and software vendors and designers. Study any predecessor systems. If system safety is a concern, study the potential hazards created by the system, and the possible ways of mitigating those hazards.

Preliminary Design

Search the internet for existing code that might contribute to the project. Figure out if and how the discovered code can contribute. Describe how the hardware, adopted software modules, and created software modules will work together to accomplish the goals of the system. Incorporate that knowledge into an architecture document. Revise requirements as necessary to accommodate facts discovered in the preliminary design.

Estimating

Guess the number of lines of code needed and the number of function points involved. Make separate estimates of project man hours from those estimates. Decompose the project into small sub-tasks and make another man-hour estimate by adding up guesses for the time required for each sub-task. Combine the estimates into a single best-guess estimate for project man-hours.

Tool Selection

Choose and install tools to be used in the project. Choice of tools depends on the processor(s) chosen, available software development stations, and the number and kind of different of microprocessors involved in the embedded system. Most developments will require at least a text editor and compiler or interpreted language, some kind of code loader, and a debugger.

Hardware Support

Support the persons developing the hardware by writing some test code. Use the information gained from writing the test code to improve the architecture document and assist with the detailed design of the system. In some cases, incorporate the hardware test code directly into the final system code.

Detailed Design

Describe created software modules and adapted existing software modules which will be included in the embedded software. Make a detailed design document that describes the software modules and their interactions with each other and the external environment. Revise requirements, architecture, and estimates as necessary to accommodate changes required to support the detailed design.

Code

Implement, in the chosen languages, all invented software modules. Integrate them with the adapted code libraries. Revise analysis, architecture, and detailed design as needed.

Debug

Make all of the hardware and software modules work as intended. Revise analysis, architecture, detailed design, and code as needed.

Integrate

Make sure the hardware and software modules interact in the way envisioned in the architecture and detailed design. Revise requirements, architecture, detailed design and code to support any changes required during integration.

Verify

Perform tests that demonstrate that the software modules work properly. Repeat the activities described above as needed.

Validate

Further test and demonstrate that the overall embedded system meets the requirements that have been specified. Repeat above activities as needed.

Chapter 4: Context

Whether you arrive at the beginning or in the middle of a project, your first task is to learn everything you can about the project. In no particular order, you need to know:

Why is this project happening?

What persons support the project?

What persons, if any, oppose the project?

Is this a new product, or a new version of an older product?

What is the technical environment of the project?

Are project resources adequate to support your efforts?

What is the expected schedule for the project? Is it realistic?

What are the skills, strengths, and weaknesses of other team members?

Which team members do you like? Which ones can you trust? What are their skill sets?

Is product safety an issue? If so, does the resource provider support an emphasis on product safety?

You may never completely answer all of these questions; but you should find out as much as you can on joining a new project.

You might say: “Look man, I’m just a programmer on this project. I do my work, they write my check. What do I care about all that political stuff?” That attitude is understandable, but ill advised. You are the one responsible for your own life. A medium sized embedded project is going to account for a big chunk of the next year or two. It’s going to affect your relationships, and your happiness for as long as you are involved. Better to know what you are walking into, than to stumble blindly into a disaster in the making.

Why is this project happening? Ask around. Normally, you will find someone who champions the project. Get to know that person. What is their motivation? If they get a spark in their eye when they talk about the project, that is a good sign. If they are just occupying an organizational position and carrying out policy, that may still be OK. If they are looking to make a lot of money, that is probably a bad sign. There are much easier ways to make money.

It can be tough at first to discover, but you need to know the primate power relationships within the organization, as they relate to the project champion. He may be on the way out, in which case you’ll have to find another situation. Make sure the project champion has the support of the main monkeys within the organization.

Check out the network of technical personnel. Have they been with the organization a long time, or did they just arrive. If the latter, are they replacing people who just left? Why? How is the morale of the technical staff? If they spend more time talking about the organization than about the work, get out of there fast.

Does your entry into the project ignite professional jealousy in anyone on the staff? If so, acquaint yourself with that person. Once you get to know each other, the problem will likely go away. You will have a new friend. If that doesn't happen, keep your eye on that person.

When you are comfortable with the project motivation and personnel, you are ready to enjoy the first technical challenge: discovering the requirements.

Chapter 5: Requirements Analysis

Before you can develop your software, you need to find out what it must do. You accomplish this by filling in the external process layer of the conceptual map. This activity is usually called requirements analysis.

If you are replacing an existing system, someone may tell you: “Make it work like the Blivitt system, only make it work better”. Life is good. Just reverse engineer the Blivitt system, and you are halfway home.

But there may be disagreement about what the system should do. You might need to hold a requirements workshop with major system stakeholders, and extract expectations in all of their diversity and conflict.

A user or client representative may come forward or be supplied and offer to work with you. Accept the offer. Be aware that a supplied representative may have expectations very different from other parties to the development.

No matter how you gather the requirements, they can be documented for two purposes: to communicate to all interested parties what the system will do, and to provide verifiable descriptions of system features for architectural, design, and testing purposes. Inasmuch as these are very different purposes, it is a good plan to document requirements in two different ways. These are the use case summary, and the formal requirements list.

Sections below describe and give examples of a use case summary and a formal requirements list. Requirements gathering techniques of reverse engineering and the requirements workshop are also described. For a wealth of useful information on requirements analysis, including details on the tools described here and much more, see *Managing Software Requirements: A Unified Approach*³.

Use Case Summary

In his aging bestseller *Object-Oriented Software Engineering*, Ivar Jacobson documented a good way to describe the interactions of a software system with its external environment. He called it “use case analysis” (use is pronounced the same way as the first word in “Yous guys grab da dame.”) It is helpful for making a first cut at system requirements, and for communicating requirements to non-engineers.

First, one identifies the different agents which interact with the system. These agents Ivar called actors. An actor could be a person, another computer, an animal, a machine, or anything else interacting with the embedded system. Actors are represented graphically by the symbol:



Actor

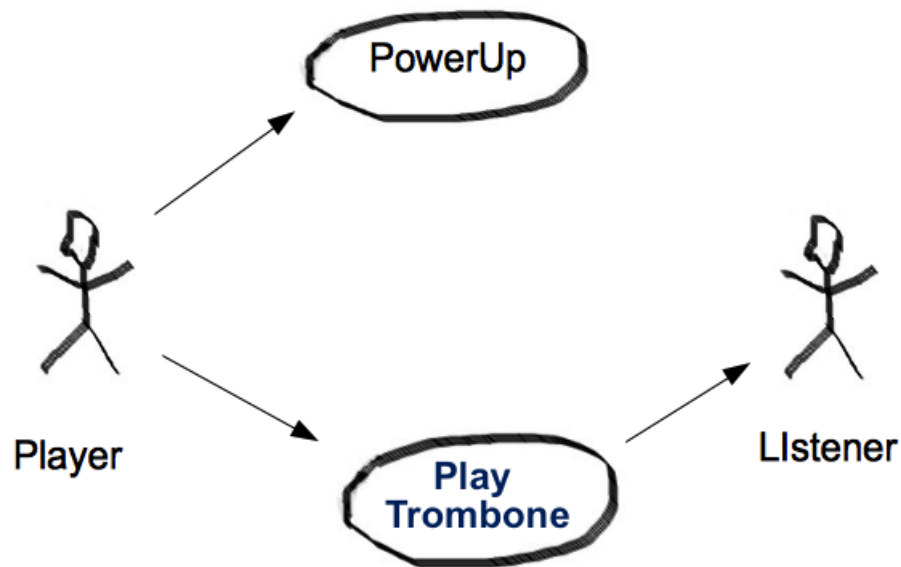
Next, one identifies a collection of situations, in which actors interact with the system. These situations, Ivar called use cases. Use cases are represented graphically by the symbol:



Use Case

In Ivar's approach, one simply lists all of the significant use cases for the embedded system, and the actors involved with each. A graphical overview of all of the use cases is optional.

Digital Trombone Use Cases



Actors

Player: The one playing the digital trombone.

Listener: Person(s) listening to the output.

Use Cases

Power Up:

The Player plugs the digital trombone into a USB socket. This causes the embedded system to go through its initialization phase.

Play Trombone:

The Player makes musical notes by blowing into the mouthpiece connected to the pressure sensor. The harder he blows, the louder the sound. The frequency of the output is changed by moving the slide. When the user stops blowing into the mouthpiece, the note ends. The Listener, who most likely will just be the Player, hears the notes played by the player.

At any time after initialization, the Player may alter the shape of the waveform produced when the Player blows into pressure sensor. When pressed, the user button selects the next waveform in a cyclical table of waveforms.

In more complex systems, there will be many more use cases. Some use cases may employ others as subroutines. See Dr. Jacobson's book for ways of diagramming many interacting use cases. As in our example, it is often sufficient to forego the graphical representation of the use cases, and to just write them as short descriptions of user interactions.

You can make a first draft of use cases for a medium sized embedded system (200-500 callable functions) in an afternoon. It may take several calendar weeks to pass the description around, hold meetings, and finally get agreement on the requirements documented by the use cases.

Formal Requirements List

A formal requirements list is a minimal, but exhaustive collection of succinct, testable statements about the operation of the embedded system as a whole. It is often helpful to include photos, diagrams or tables in the formal requirements list.

Sometimes the formal software requirements are placed into their own document. Sometimes they are included in a separate section of the formal system requirements. Sometimes they are tagged as software requirements within the formal requirements document. Sometimes they are kept in a requirements database or spreadsheet.

Why do you need a formal requirements list?

- It keeps all of the team members working toward the same goals.

- It will guide you during the preliminary and detailed design.

- It will help you in making tradeoffs.

- You will use it to write test plans that measure how well your work turned out.

The process of writing it will expose you to important issues you might not otherwise discover.

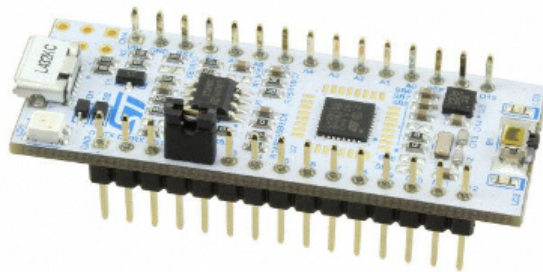
There are many recommended formats for requirements lists. There's an IEEE standard. There are U.S. government standards. There may be standards within the organization which is sponsoring your project. Don't fight over the format. The only thing you need is a collection of sentences, tables, and/or diagrams, each of which makes a succinct, testable statement about the operation of the software. That can be put into any format necessary within context of your project.

In no event should you attempt to communicate requirements to non-engineers using the formal requirements list.

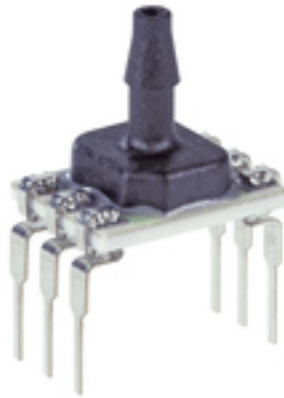
Digital Trombone Formal Requirements

The hardware for the Trombone project has already been chosen, primarily for its ready availability, low price, and compliance with de-facto standards.

1. The digital trombone shall use a NUCLEO-L432KC, mbed-Enabled Development Nucleo STM32L4 MCU 32-Bit ARM® Cortex®-M4 Embedded Evaluation Board. This is an Arduino NANO compatible, Arm Cortex-M4 processor with minimal supporting hardware.



2. The Trombone accepts input from a Honeywell Pressure Sensor, part #ABPDANN005PG2A3, desc: DIP, barbed, 5 PSIG.



3. Trombone slide shall be a 10 centimeter linear slide potentiometer.



4. The User Button will be a two contact, spring-loaded-to-off push button.



5. A headphone jack will be supplied such as the one shown below:



6. Within two seconds after power-up, the Trombone enters normal operation with a saw waveform.
7. Every note played consists of a waveform modulated by a pressure envelope.
8. The waveform begins at the frequency determined by the potentiometer position, and a volume determined from the pressure sensed.
9. Individual notes begin when pressure is applied to the pressure sensor, and exponentially decay when it is removed. Between application and removal of pressure, the volume and frequency may vary with pressure and slide position changes.
10. The digital trombone will support the same frequency range as a basic tenor trombone (E2 to A5).
11. The trombone reacts to a press of the user switch by choosing the next waveform in the circular waveform table for the subsequent note or notes. Waveforms supplied include Saw, Square, Triangle, and Sine.
12. The waveform may change when no note is currently playing, or in the middle of a note.
13. No cracks or pops will be heard except possibly when the waveform is changed in the middle of a note.

Reverse Engineering

If it is necessary to reverse engineer an existing system to jumpstart the requirements definition, there are several alternative ways to proceed:

The easiest way to obtain the requirements is to get a copy of the requirements document for the previous embedded system. Update it with any new or changed requirements, and you are done. This sounds good, and to the extent it is possible, should provide the first cut at the new system's requirements. Unfortunately it doesn't always work. There may not have been an original requirements document; or the feature set changed over the life of the product, and the requirements document may not have been updated.

If your need for accurate requirements is not met by the first alternative, obtain the user's manual of the previous system. This is often a good source of requirements data. It can be supplemented by actually using the previous system, and observing what it does.

If a previous requirements document, and the previous user's manual are unavailable (real engineers don't write documents), or don't give you enough information; try to

find architecture and design documentation for the previous system. If you find such information, it may be sufficient to infer both use cases and a formal requirement list. Read through all of the user, architecture, and design documents you have found, and attempt to create use cases first, and then the requirements list.

If there was no previous architecture and system documentation, or if it was inadequate to produce the use cases and/or the requirements; you must study the source code of the previous system. Even if the documentation allowed you to construct the use cases, it may not have been detailed enough to help you with the detailed requirements list. For that you may still need the source code.

One thing you can do with source code is to make call trees for every interrupt and every task you discover in the source code. Interrupts tend to handle functionality associated with the virtual machine layer. Tasks tend to handle functionality associated with the application layer. Carry the call trees down to the lowest level of functions that don't call any other functions. You will find useful details even at that level. The source code and call trees should give you enough information to write the use cases and the detailed requirements list. If they do not, it could only be because the source code is so hard to read that you cannot decipher it. If that is the case, give up on reverse engineering and find another way to obtain requirements.

Requirements Workshop

When creating an entirely new system, or when you are unable to reverse engineer the previous system, it will be necessary to hold a series of meetings with persons who know how the new system must work. For many, the most enjoyable way to do this is with the requirements workshop.

The requirements workshop is a one or two day meeting (duration depends upon size of the new system), between the system developers and resource providers or stakeholders. There are two goals. The first is to introduce the stakeholders and developers who will be working together. The second goal is to familiarize the developers with the stakeholder's requirements for the new system.

By working and eating together for a day or two, the participants will naturally get to know each other, and thus meet the first goal of the workshop. The second goal, transmission of the requirements, will take place during the work sessions.

The work sessions should include the following:

Introductions - Introduce participants, lay ground rules, icebreaker exercise, agenda for the workshop.

Historical roots - Answers the questions: How is the job of the new system currently done? This is a presentation by one or more system stakeholders. It will hopefully include a visit to an actual work site.

System operational environment - Presentation of the environment in which the new system will operate. This should include the organizational environment, the physical

environment, the regulatory environment, the sales environment, and the maintenance environment.

System development environment - The developers give an overview of their own facilities, including a map to the location, phone numbers of key people, persons assigned to the project, their backgrounds, and key equipment that may be used on the project. The stakeholders describe the persons involved in their management, engineering, their background, and contact information.

System goals and constraints - All participants cooperate in creating a prioritized list of each of the different things the new system must do or be, and each of the things the system must not do or be.

Future activities - The participants agree who will produce use cases and a formal requirements list, and when they will meet to review them. This workshop should be sufficient to jump-start cooperation between a team of developers and a group of stakeholders in a new embedded system project.

In case the developers and stakeholders work for the same company, the agenda can be somewhat abbreviated, especially if both groups are co-located. See Chapter 10 of ***Managing Software Requirements: A Unified Approach***³ for a detailed discussion of the requirements workshop.

Requirements Model

Some developers, after they have the use cases and the requirements list, construct a so-called requirements model of the system. The requirements model is a first attempt to structure the application layer. It defines an interactive collection of virtual objects that meets the system requirements, and is, at least conceptually, capable of running on an idealized virtual machine.

Early requirements models used a data-flow virtual machine. Jacobson² proposed an, Interface-Entity-Control virtual machine for the requirements model. While it helps to construct a requirements model, it is not always necessary.

If you do an architectural study like the one shown in the next chapter, you can document a collection of virtual objects that are compatible with the virtual machine and hardware layers of your target system.

Safety Issues

Embedded systems are often used in circumstances in which system failure endangers life or property. This is particularly true of military, and medical systems. It is also a consideration in automotive, laboratory, industrial, and consumer systems.

An important part of requirements analysis is determining to what extent the proposed system is hazardous to life or property, and what may be done to mitigate the hazards. Medical and military

development environments have long employed standard procedures to build safety into embedded systems.

The hazard analysis is one of the procedures used to focus attention on safety issues during analysis. Create a list of all the hazards the new system may present to the developer, user, maintainer, or passersby. List the severity of each potential hazard, and give some idea of its probability of occurrence. The risk associated with each hazard is the product of its severity times its probability of occurrence. Suggest methods for mitigating the riskiest hazards, through provisions in the hardware, the software, or documented procedures for using the system.

The hazard analysis may be the only safety-related thing you do during the analysis part of the development, but it will help to focus attention on safety issues early in the project. Hazard analysis is included in the requirements work because it may lead to additional requirements for the system. For more information on hazard analysis, start with https://en.wikipedia.org/wiki/Hazard_analysis

Chapter 6: Preliminary Design

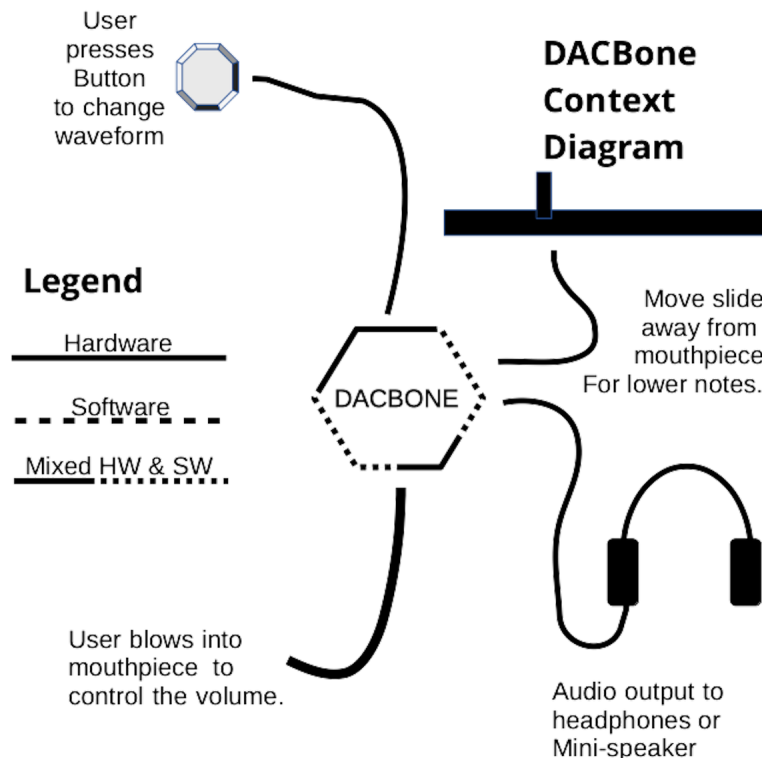
The preliminary design answers the question: “What are the major hardware and software components of the embedded system, and how do they interact to meet the system requirements?” This question can be answered by producing an **architecture model** of the new system.

If there is not yet enough information to produce an architecture model, more time must be spent gathering information about the hardware objects of the system, further developing the requirements, and gathering source code and design information about systems with similar hardware and requirements. It may also be helpful to hold an architecture workshop with other engineers on the project.

In this chapter, an architectural model is developed for our example of the digital trombone.

Digital Trombone Architecture

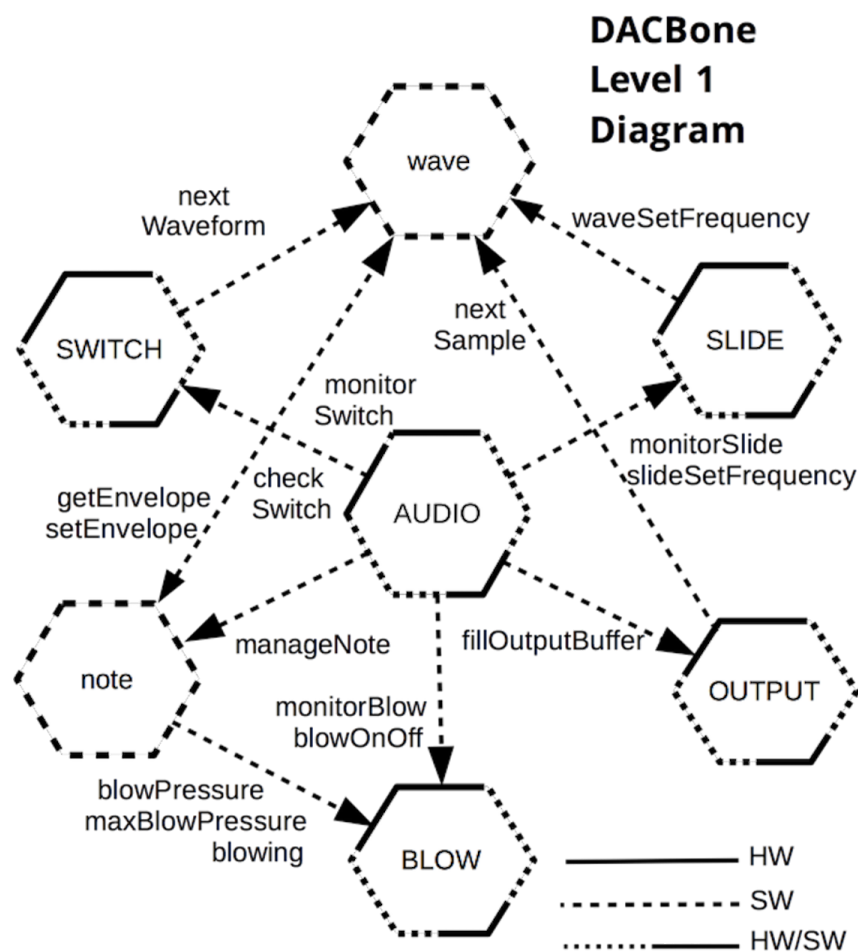
The architecture model begins with a system context diagram. This diagram represents the embedded system as a single symbol, and shows its interactions with the external environment.



User touch controls the frequency of musical notes. Volume of the notes is controlled by blowing into the pressure sensor. The user switch controls the waveform to use for the next note.

Level 1 Diagram

The Level 1 diagram breaks down the single object, into a group of objects, with the interactions between them. Software objects and influences are shown with dotted lines. Hardware objects and physical influences are shown with continuous lines. Mixed hardware/software objects and influences are shown with lines having both dotted and continuous segments. Data Flow may sometimes be indicated by short lines with a circle on one end and an arrowhead on the other.



The BLOW object periodically reads the pressure sensor, updates the current pressure value, and keeps track of whether the user is blowing or not blowing into the mouthpiece.

The SLIDE object is periodically asked by the AUDIO object to compute the current frequency from the slide position, and pass that frequency to the **wave** module.

The AUDIO object coordinates the activities of the SWITCH object, the SLIDE object, the OUTPUT

object, the BLOW object and the **note** module to create the sounds expected by the user when he blows into the pressure tube, presses the user switch, or adjusts the slide.

The **wave** module computes each buffer that is passed to the DAC DMA based upon its latest frequency, previous envelope value, ending envelope value or release ratio, and the most recent choice of waveform. It also receives nextWaveform requests from the SWITCH object, waveSet-Frequency calls from the SLIDE object and setEnvelope calls from the **note** module. Whenever the phase variable exceeds the value corresponding to 2π , *it resets the phase value to the excess above the 2π value.*

The **note** module keeps track of whether the user is blowing into the pressure sensor or not, and computes an envelope value when the user is blowing. When the user is not blowing, it monitors the decaying envelope produced by the **wave** module, and enters a NO_NOTE state when the decaying envelope reaches practical zero.

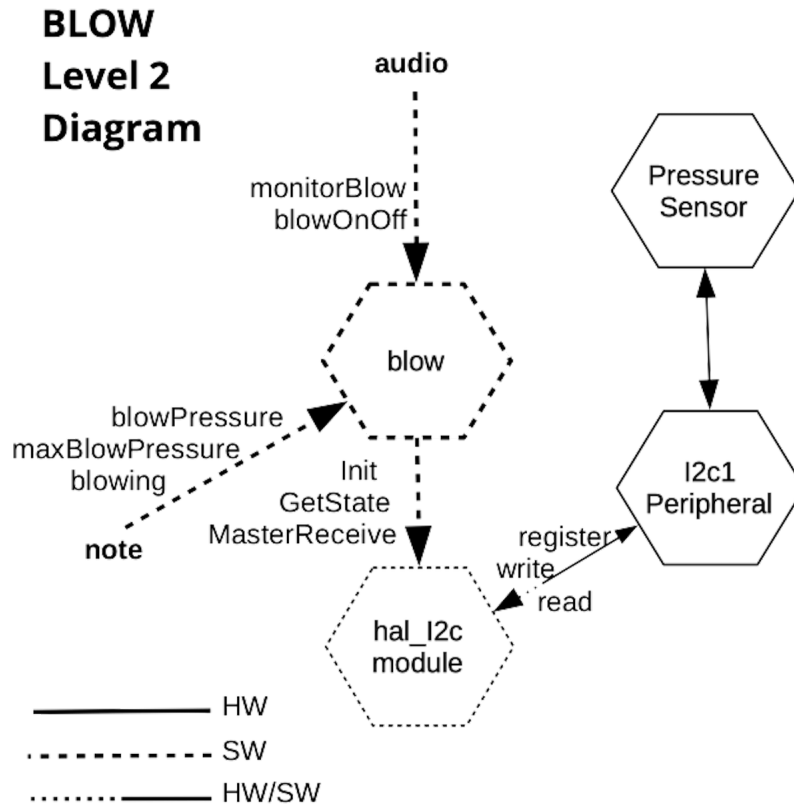
The **SWITCH** object watches the user switch, and tells the **wave** module to move to the next waveform when the switch is pressed.

The **OUTPUT** object receives fillOutputBuffer function calls from the AUDIO object as fast as the system main loop can call the AUDIO object manageAudio function. If the fillOutputBuffer function determines it is time to refill half of the DMA buffer, it repeatedly calls the **wave** module nextSample function, placing the received values into the DMA buffer which feeds the OUTPUT object DAC peripheral.

Level 2 Diagrams

These diagrams show the internal structure of the mixed objects or composite objects described in the level 1 Diagram.

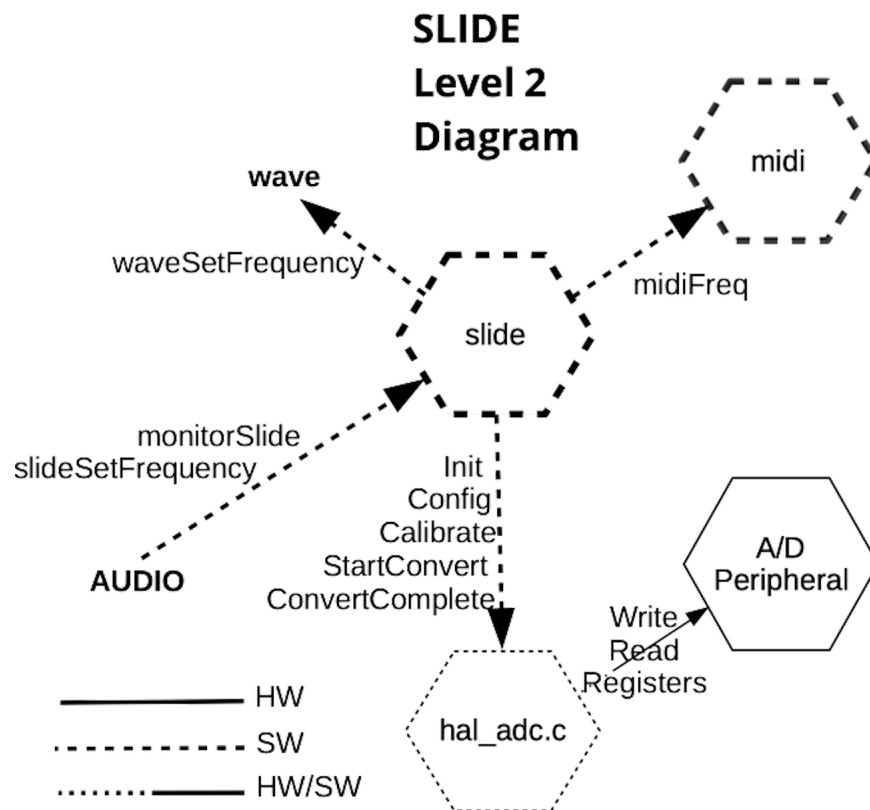
The **BLOW** object consists of the **blow** software module, an i2c firmware library module, an MCU i2c peripheral, the Honeywell pressure sensor, and the mouthpiece into which the user blows.



All hal module names are prefixed by stm32l4xx_

The **blow** module uses the i2c library module and the on-chip i2c peripheral, to setup and periodically read the raw pressure at the pressure sensor, record whether or not the user is blowing into the tube, and return the raw pressure, blowing state, and ratio of raw to maximum pressure when it is requested by the **note** module.

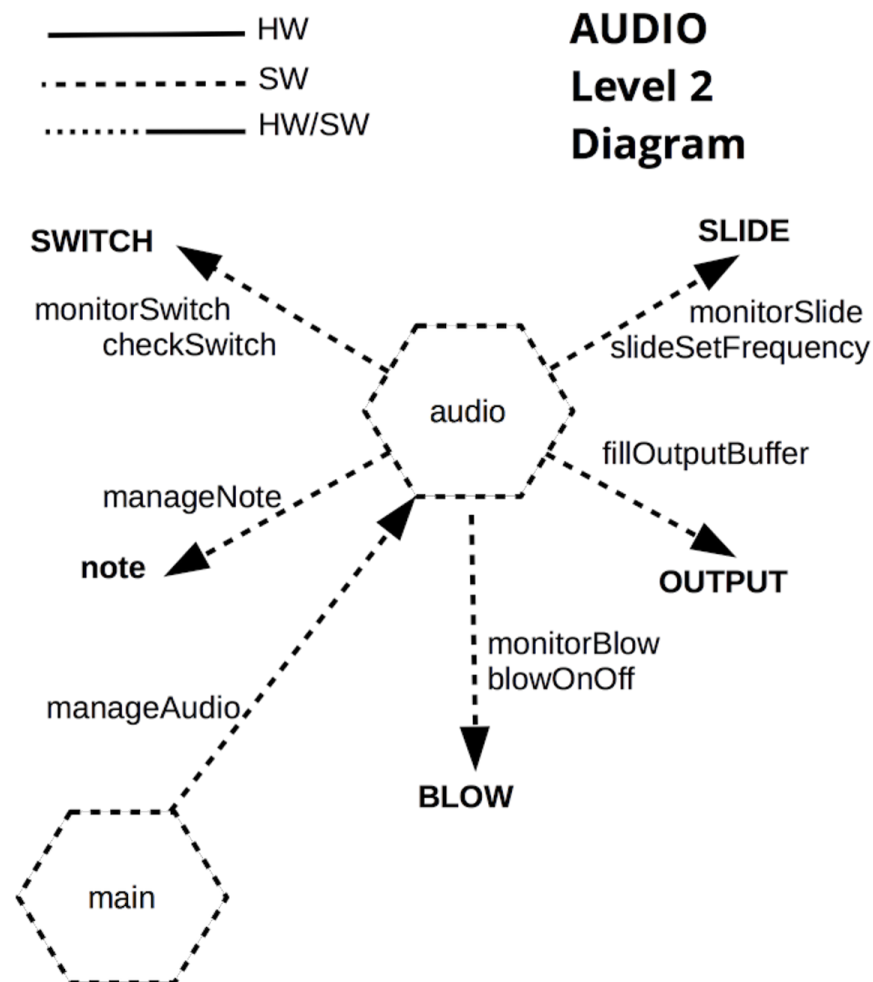
The **SLIDE** object consists of the **slide** software module, a **midi** software module, an A2D library module, and an MCU A2D peripheral.



All hal module names are prefixed by `stm32l4xx_`

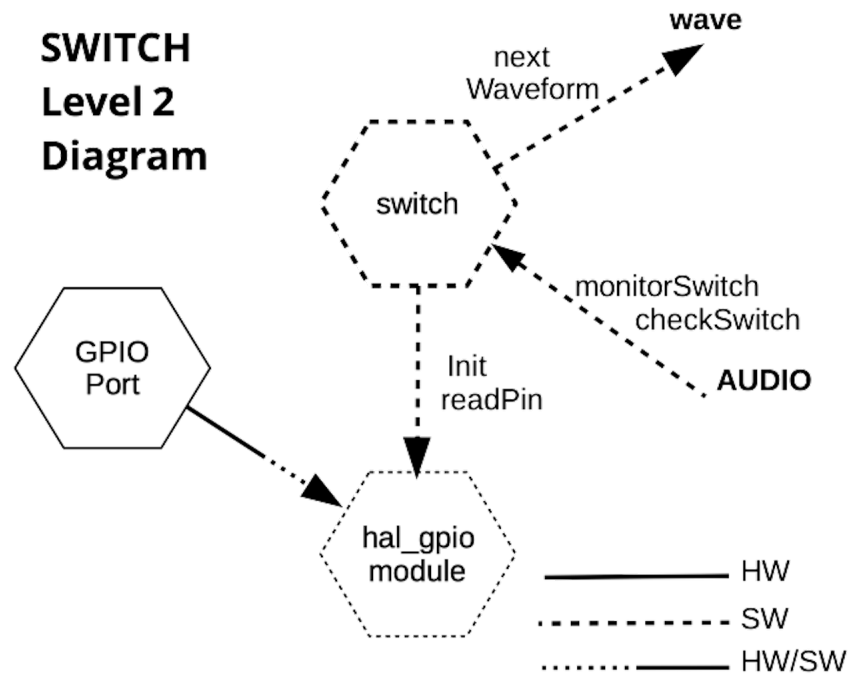
The **slide** module uses the A2D library module and on-chip A/D peripheral to read the current voltage across the linear potentiometer, and the **midi** software module to compute the current frequency for use by the **wave** module.

The **AUDIO** object consists of the **main** software module, and the **audio** software module. It has a `manageAudio` function which is called repeatedly by the program main loop.



The `manageAudio` method calls `monitorSwitch`, `monitorSlide`, and `monitorBlow` methods of their respective modules to update the current waveform, frequency, and whether the user is blowing, and if so, how hard. Then it checks to see if it is time to refill a recently output buffer-half, and if so does it, and then calls the **audio** module `updateParams` function, saves the current values of blowing state, switch state, and frequency to use on the next buffer-half. It also calls `manageNote` in the **note** module.

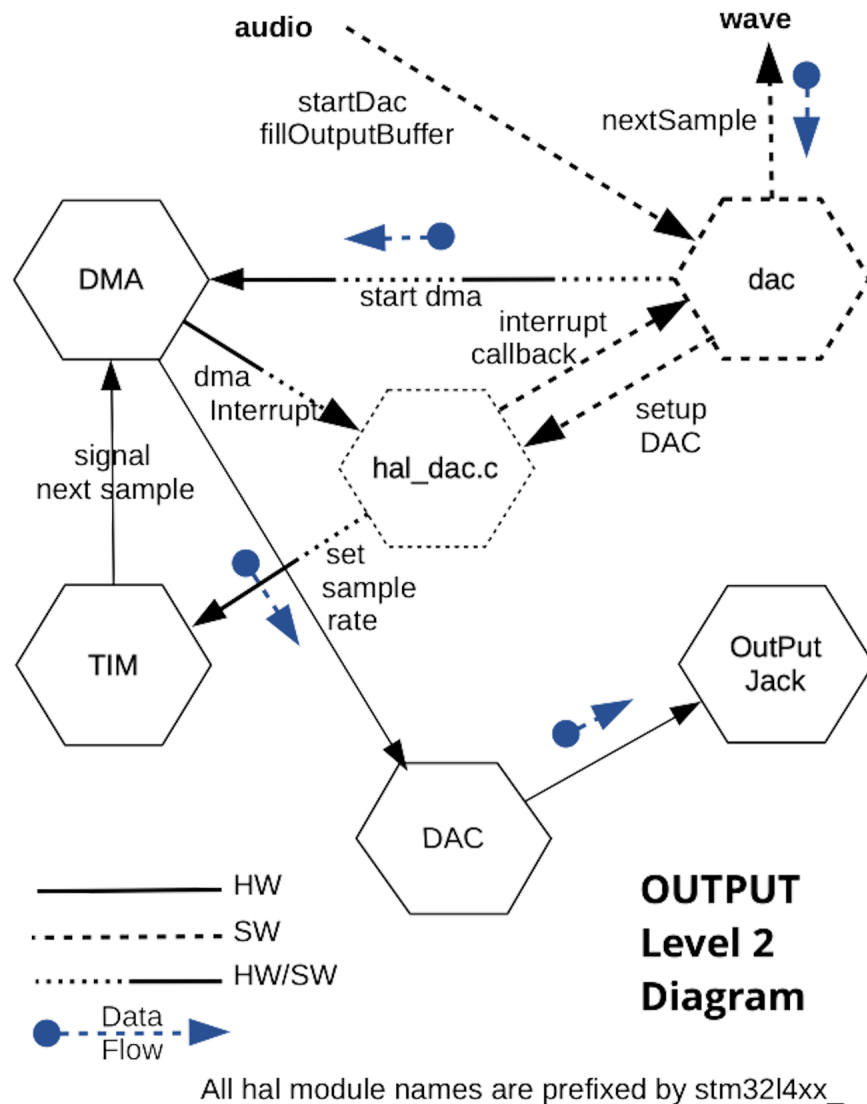
The **SWITCH** object consists of the **switch** software module, the GPIO library module, and the GPIO Port peripheral.



All hal module names are prefixed by stm32l4xx_

It has a `monitorSwitch` function which debounces the switch and keeps track of when the switch opens or closes. The `checkSwitch` method calls the **wave** module `nextWaveform` method when the switch has been closed for longer than a threshold value. After that the switch must be re-opened before another call can be made to `nextWaveform`.

The **OUTPUT** object consists of the **dac** software module, the dac library module, and DMA, DAC, and TIM peripherals, as well as an output jack.



The `fillOutputBuffer` method of the **dac** software module checks to see if the DAC triggered DMA transfer of a filled output buffer-half has completed. If so, it refills that half of the **dac** circular output buffer by repeatedly calling the **wave** module `nextSample` method, and placing the returned samples into the recently output buffer-half until it is filled.

Once started by the **dac** module `startDac` method, the **TIM** timer continually triggers the **DMA** to transfer the next sample to the **DAC**, and the **DMA** continually requests buffer refills via the transfer half complete and transfer complete interrupts. When the `fillOutputBuffer` finishes filling a buffer half, it returns a code notifying the **AUDIO** object that it has done so. This tells the **AUDIO** object to call the routines which update audio parameters.

The diagrams above were produced by trial and error while studying information on the hardware components of the digital trombone and thinking about how to structure the software that uses them. Then, as working code was developed, they were updated to reflect the final form of the software architecture. For larger projects, it is often helpful to hold a workshop for the project engineers to collaborate on a set of architecture diagrams.

Architecture Workshop

It may sound strange, but a group of electrical, mechanical, and software engineers will typically enjoy a morning spent functionally decomposing an embedded system. I have seen groups of engineers take over an architectural workshop, and produce dozens of diagrams of the different functional aspects and levels of an embedded system... and enjoy it. Engineers like clarity. An architectural workshop brings clarity to their understanding of the relationships between the components of the embedded system.

It's easy to start an architecture workshop. You just make sure that the engineers show up at the appointed time and place. Bring a flip chart and some markers. Introduce the engineers to the concepts of hardware, software, compound and mixed objects, and lead them through the creation of the system context diagram. Next you help them with the Level 1 diagram. By the time they have finished level 1, the engineers are ready to take over, each of them leading the decomposition of a piece of the system that appears on the Level 1 diagram.

It is a good idea to have worked out in your own mind what the Level 1 diagram should look like before you help others produce it. By the time everyone runs out of gas, you have two things: a whole bunch of diagrams of the embedded system architecture, and enhanced clarity in the minds of everyone present. Take the diagrams and put them into an architecture document. The clarity will pay off in everybody's work for the rest of the project.

For the rest of the project, the diagrams will remind people of the agreed upon architecture, and provide focus for discussions of necessary tradeoffs or changes.

Chapter 7: Estimating

Estimating is a skill that only grows with practice. Always do a project estimate when the preliminary design is done. Over time your methods and results will improve. Produce your estimate using three different techniques. I use the Cocomo, Modified Function Point, and Add-Up-the-Guesses techniques. Once you have your three estimates, combine them in whatever way your conscience, your bank account, and your creativity dictate. Don't worry about being wrong. You probably will be, but if you give it your best effort; you will not suffer unduly when your estimate is later proved incorrect. Of course, if you have your own money riding on the outcome, it helps to be right.

COCOMO Method

Guess the number of lines of code in your project. Then make swags at the values of a collection of parameters which describe the project. Finally, apply the COCOMO formulae to compute the total effort, calendar time, and number of programmers necessary to accomplish the mission. For details, refer to the book: *Software Cost Estimation with COCOMO II*⁴.

For our digital Trombone example, the modules to be coded are main, audio, blow, note, slide, switch, midi, wave, and dac. New lines of code guesses for the modules to be written are:

module	lines
main	22
audio	20
blow	40
note	25
slide	30
switch	30
midi	40
wave	120
dac	80

(The modules will doubtless incorporate example code supplied by the chip and board vendors as well as complete modules supplied by a hardware access layer. These lines of code are not counted in the total lines to be written).

I guessed 287 lines of code, not counting libraries already written by chip and board maker, and for parameters, picked:

parameter	setting
mode	organic
analyst capability	high
application experience	high
complexity	nominal
database size	low
language experience	high
programmer capability	high
required reliability	nominal
requirements changes	nominal
schedule pressures	low
main storage constraint	low
execution time constraint	high
use of software tools	high
computer turnaround time	low
virtual machine experience	nominal
virtual machine volatility	nominal

Executing the magic formulae (see Appendix A) returned the following values:

- Total Man months 0.81
- Total Calendar months 2.3
- Developers Required 0.34

Since I planned to work half time instead of 1/3 time, I recomputed the calendar months as $0.81 / 0.5 = 1.62$.

Modified Function Point Method

In the normal function point method, you add up the number of inputs, outputs, transaction types, intermediate file types, external file types, and so on to arrive at the number of function points. For embedded systems, you can just add up the number of non-redundant influences and objects in your architectural diagrams. If you don't have any architecture diagrams, make some. Use that count as the number of function points.

Then you apply a collection of formulae to obtain a variety of results. For the digital Trombone device example, the number of function points came out to be 58. Using formulas in a script program sent to me (thanks Kevin), the estimate came out to:

computed result	numeric value
Function Points	47
Paper pages deliverable	84
Word count	33494
Function Pt Growth when done	4.75%
Critical creep	115 %
Test Cases	102
test runs	406
Defect Potential	123
Build Staffing	0.31
Schedule (months)	4.66
Maintenance Staff	0.09
Total Effort	1.46 man months

Since I planned to work half time instead of 0.31 time, I recomputed the calendar months as $1.46 / 0.5 = 2.43$.

The substitution of object/influence count for function points was arbitrary; but the results often agree pretty well with other estimates. If you want to use function point analysis in a more formal way, see the book *Function Point Analysis*⁵.

Add Up The Guesses Method

In this method, you write down everything you are going to do and guess how long each action will take, for example:

Task	hours
Figure out why I'm doing this	1
Do use cases	1
Do architecture diagrams	10
Estimate project effort	3
Research algorithms	10
Initial Class (Module) diagrams	10
Steady State Interaction Diagram	6
Initialization Interaction Diagram	1
Module Descriptions	8
Set up development environment	1
Write hardware and tool test program	1
Setup IDE and makefile	8
Code and debug main module	8
Code and debug audio module	8
Code and debug blow module	12
Code and debug switch module	8
Code and debug note module	8
Code and debug slide module	15
Code and debug midi module	4

Task	hours
Code and debug wave	20
Code and debug dac	10
Integrate, test, and revise	25
Devise tests to verify algorithms	16
Devise test to demonstrate it works	4

- Total Hours 198
- Man Months 1.24
- Calendar Months = Man Months * 2 = 2.48

(The factor of 2 in Calendar months is because I expect to work only half-time on the project.)

Combining The Estimates

A weighted average often works OK here.

Then the problem of combining the estimates reduces to one of choosing the weights. You weigh more heavily those estimates that conform to your gut feeling about the project. Then you find some convenient way to rationalize that decision. It is perfectly OK, and may be wise, for the weights to add up to a number substantially bigger than 1.

With equal weights for each estimate, we get:

$$(1.62 + 2.43 + 2.48)/3 = 2.18 \text{ calendar months}$$

Just looking at the three estimates and adding a bit to be safe, it looks like the digital Trombone device is going to take somewhere in the neighborhood of 2.5 calendar months to complete. It will require roughly half of my time during those two and a half months.

Appendix A contains cellular calculator scripts for the COCOMOII and Function Point calculations.

Chapter 8: Tool Selection

For the digital Trombone project, there are many toolsets available, including IAR, Keil, GCC-toolsets, the ST version of the Atollic IDE, and Arm Mbed. Visual Studio Code (VSCode) and the embedded gcc Arm Toolset were chosen for editing, compiling, and debugging. The tools were configured as described in the companion eBook Cross-Platform Cortex-M Development.

The STMicroelectronics STM32CubeL4 chip library Hardware Access Layer (HAL) was used for accessing the chip features. Included example code was mined for snippets used in peripheral access and other functions.

There is no need for an RTOS. Even when there are several parallel processes, their activities can be better coordinated by devoting a state machine or event chain to each process, rather than incurring the complexity, overhead, and interrupt latency issues of an RTOS.

There is a USB port on one end of the board, which connects to an onboard STLinkV2-1 SWD interface that communicates with the background mode debugger on the processor chip. Driver support, if needed, is available at <https://www.st.com/en/evaluation-tools/nucleo-l432kc.html#sw-tools-scroll>. An STLink Server is available at that location, but for the purposes of this eBook, openocd is used. The openocd scripts folder will need a board script called stm32l432kc_nucleo.cfg the text of which is shown below:

```
# This is a ST NUCLEO L432KC board with a single STM32L432KCU6 chip.
# This is for using the onboard STLINK/V2
source [find interface/stlink-v2-1.cfg]

transport select hla_swd

# increase working area to 48KB
set WORKAREASIZE 0x0c000

# chip name
set CHIPNAME STM32L432

source [find target/stm32l4x.cfg]

# use hardware reset
reset_config srst_only srst_nogate
```

This nucleo board is both mbed and Arduino compatible. Code can be loaded by dragging the compiled .bin file to a mass storage device that appears on the desktop whenever the processor board is connected to the PC via USB.

The USB connection includes the ability to direct printed output from the running program to a remote terminal program running on the PC. So some debugging can be done by printing information to the terminal on the PC. For some reason, that did not work on my MacBook Pro, so I made do with breakpoints and watched variables.

The board can also be programmed with the arduino development environment, but it was not used for this project.

VSCode source folders and makefiles for Linux, Mac, and Windows, are supplied as extras with this eBook for building and loading the code with the embedded arm gcc toolset, and debugging with the VSCode debugger and the openocd server.

For more information on configuring your computer for the VSCode development environment, you can find it in the companion eBook: Cross-Platform-Cortex-M-Development which is bundled with this eBook at leanpub.com.

A logic analyser can be used to debug the I2C connection between the nucleo board and the Honeywell pressure sensor. The Saleae USB Logic Analyzer available from SparkFun is a good 8-bit logic analyzer, and it worked nicely for the digital Trombone project.

Chapter 9: Hardware Support

Often, an electrical engineer develops a printed circuit board while you are writing requirements and doing the preliminary design. A mechanical engineer may simultaneously develop mechanical components.

The hardware engineers can be helpful with things like getting the right power supply for the prototype boards, suggesting the right algorithm for custom hardware access, making sure the processor is running, setting up processor clock chains, providing standoffs for the circuit board to keep it off of the desktop, assembling and disassembling cases, and supplying the history of the product.

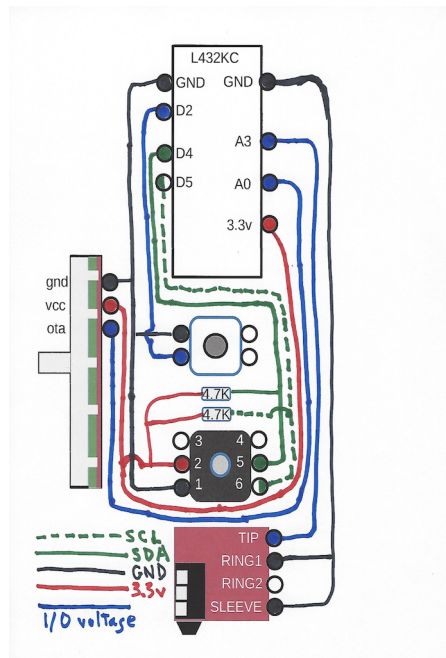
In return, they will expect you to provide early versions of software to test board features, light blinking programs to test visibility of LEDs, and sometimes to help debug mystifying problems with the circuit board.

The project is not going anywhere until you get production circuit boards. Since there is frequently a long lead time on circuit boards, you must test as many program features as possible on an eval board or the prototypes. This helps to verify some hardware design features, and gives you confidence working with and changing the software design. You will likely make quite a few such changes in the course of the project.

A good relationship with the hardware engineers is not only vital to the success of the project. It is vital to your own success. Fortunately, most hardware engineers are pretty easy to work with. They have had most of their rough edges rubbed off by contact with reality. There are notable exceptions to that rule.

In the digital Trombone project, the processor, application, and peripheral boards were purchased off the shelf. There was no need to support debugging those boards. However it was necessary to configure and make the connections between the pressure sensor, the slide, the switch, and the processor board.

The components were mounted on a breadboard, and jumpers were used to connect them in accordance with the diagram shown below:



Chapter 10: Detailed Design

The detailed design chooses a **run-time pattern** and provides a **module overview diagram** of the software modules in the system. Each module entry in the overview diagram summarizes the essential data structures, language objects, functions and methods of that module.

The detailed design also includes an **module interaction diagram** for each use case. This diagram shows the modules involved with the use case, as well as all of their method and function calls employed in that use case.

A **module catalog** is produced which provides the information needed to write the code for each module in the system. Detailed information is given on each data structure or language object, and each method or function call named in the overview diagram for each software module. Algorithms, state diagrams, and flowcharts may be included as needed, as well as suggestions for testing each module.

Runtime Pattern for Digital Trombone

The runtime pattern describes the decisions made about how the software modules will interact in time. It answers the questions:

How many processes must be handled simultaneously?

How many interrupts will be employed?

What techniques will be used to manage those processes and interrupts?

The separate processes handled by the digital trombone include:

user button pushes

user slide manipulation

user mouthpiece pressure changes

musical note frequency, attack, sustain, decay and release.

Since the ST ARM processor system clock will be running at 80 MHz, it should easily handle these processes with a single task which calls separate functions to incrementally manage each process. Some of the process management functions may need to be state machines.

After initialization, DMA complete and half-complete interrupts will trigger interrupts that refill, with audio samples, alternate halves of the circular DMA/DAC buffers. TIM interrupts will signal the DAC object when to read and convert to analog the next value in the current DMA/DAC buffer.

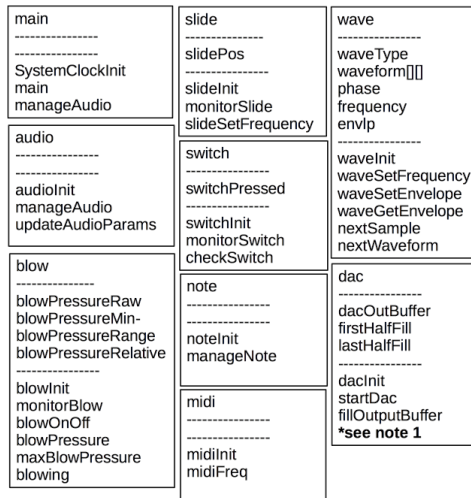
Module Overview Diagram for Digital Trombone

During the design, it is a good plan to keep an up-to-date diagram of all the software modules in the system, and for the newly created modules, the names of their essential data structures, language objects, and the function and methods they provide.

The diagram can be begun early in the detailed design phase, using the software modules and interactions shown in the architecture model of the preliminary design.

The module overview diagram will be revised occasionally as the detailed design and coding progress.

The final module overview diagram for the digital trombone is shown below:



note 1: HAL DAC Interrupt callback functions with names too long to include in box.

HAL_DAC_ConvHalfCpltCallbackCh1(DAC_HandleTypeDef *hdac)

HAL_DAC_ConvCpltCallbackCh1(DAC_HandleTypeDef *hdac)

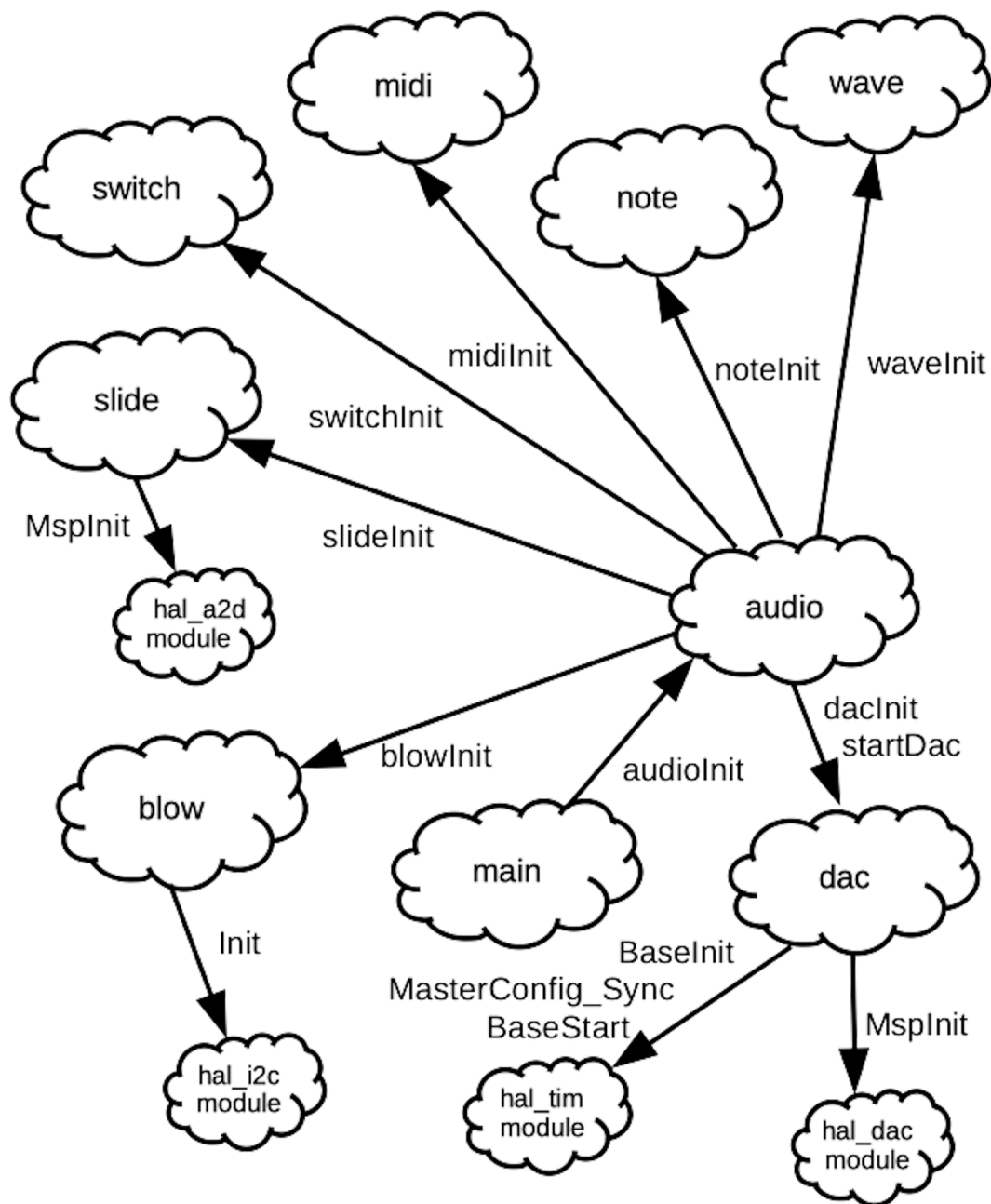
Override their weak namesakes in the HAL.

Module Interaction Diagrams for Digital Trombone

A module interaction diagram shows the interactions between software modules to accomplish a specific goal of the system, such as power-up, or a single use case. The module interaction diagrams may be started by copying software module interactions from the architecture model in the preliminary design.

The interaction diagrams should be updated regularly and tested frequently for compatibility with the other design diagrams as development progresses. Inconsistencies between diagrams should be resolved as they are noticed.

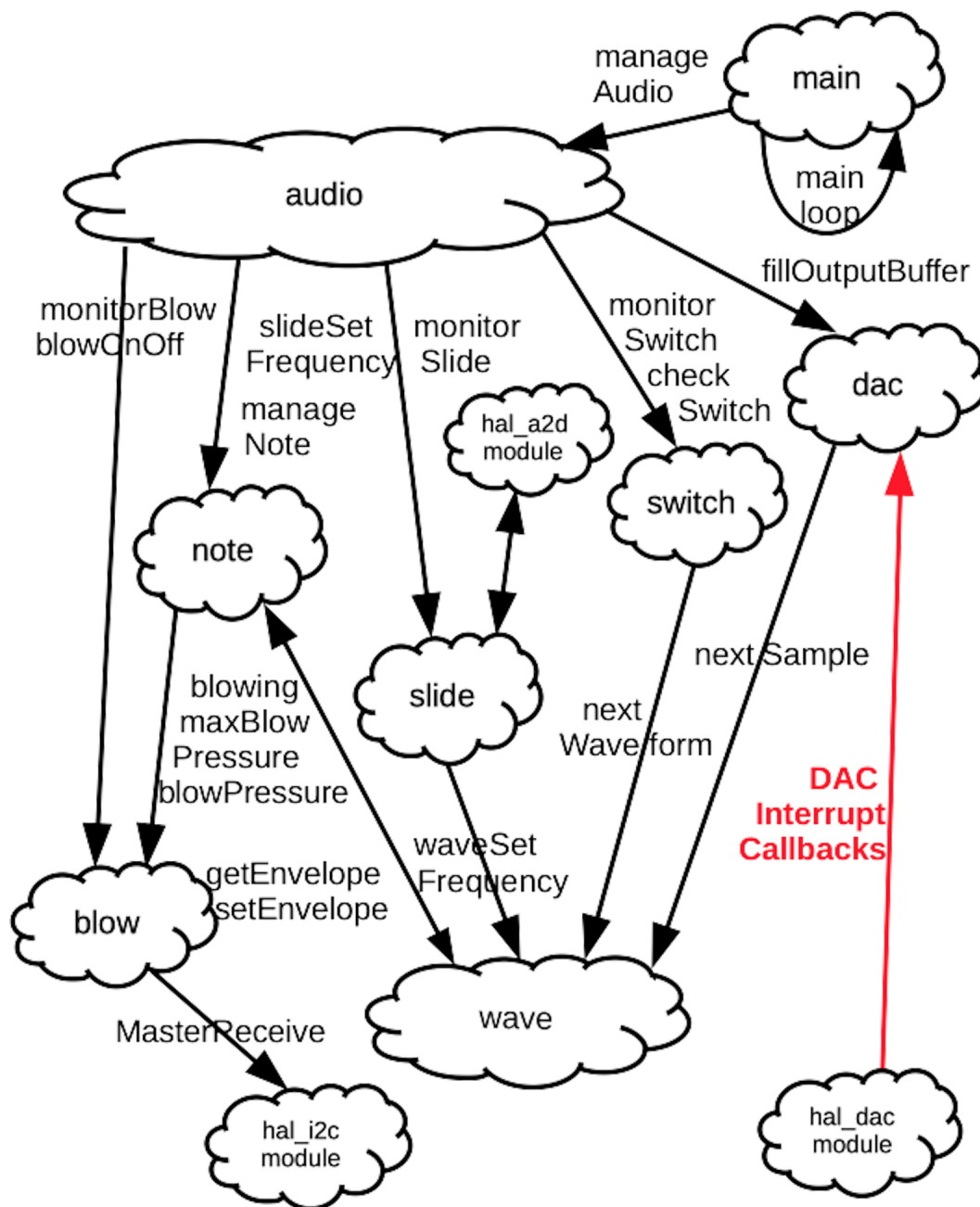
The first interaction diagram shows the module interactions for the Power-up use case.



Object Interaction Diagram for PowerUp Use Case

Each module used in the power-up use case is shown as a cloud. The arrows are function or method calls made from one module to another.

Each arrow is labelled with the name of the function or method called in the target module. This diagram shows the power-up initialization call tree, executed before the main loop is entered.



Module Interaction Diagram for PlayTrombone Use Case

Nearly all the method calls in the Play Trombone use case take place under the control of the main loop. Those calls are indicated by black arrows with black labels. Two of the method calls take place under the control of the DMA DAC interrupt, which is triggered by the DMA assigned to the DAC in the dac library code. That interrupt is the DAC interrupt which is shown with a red arrow and label.

The main loop continuously calls only one method, the manageAudio method of the audio module. The manageAudio method calls four methods every time it is called: monitorBlow, monitorSlide,

monitorSwitch and fillOutputBuffer.

The monitorBlow method of the blow module is a state machine which obtains readings from the pressure sensor. It uses them to compute the current relative pressure on the pressure sensor. The audio module also calls the blow module blowOnOff, whenever it has filled an outputBuffer, to update its estimate of whether the user is blowing into the pressure tube.

The monitorSlide method of the slide module is a state machine which reads the analog digital converter assigned to the slide potentiometer to determine the current slide position. Once it determines the slide position, the state machine calls midiFreq in the midi module to find the closest midi note corresponding to the slide position, and calls the wave module waveSetFrequency method to update the current wave frequency value, which is used to compute the frequency for the next output buffer.

The monitorSwitch method of the switch module debounces the user waveform switch. The checkSwitch method is called by the audio module from its updateParams method. If the switch has been pressed since the last buffer was filled, it calls the nextWaveform method of the wave module.

The fillOutputBuffer method in the dac module sets a flag whenever the DAC interrupt calls one of the two buffer half sent callbacks, bufferHalfComplete and bufferComplete. Then it returns non-zero to manageAudio, indicating it has filled a buffer. When fillOutputBuffer returns non-zero, manageAudio also calls one additional method, updateAudioParams.

When the updateAudioParams method is called from manageAudio, it calls an additional four methods: slideSetFrequency, blowOnOff, checkSwitch, and manageNote. These methods update audio parameters to use the next time fillOutputBuffer fills a buffer.

Only one DAC buffer is needed by the DAC DMA interrupt. When the first half of the buffer has been transferred, an interrupt routine is invoked by the DMA assigned to that DAC. The interrupt routine calls the **dac** module callback routine with the very long name *HAL_DAC_ConvHalfCpltCallbackCh1*. That routine sets a flag in the **dac** module which tells the fillOutputBuffer routine, the next time it is called from the **audio** module, to fill up the first half of the DAC buffer by repeatedly calling nextSample in the **wave** module.

When the entire DAC buffer has been transferred, another interrupt routine is invoked by the DAC DMA. That interrupt routine calls the **dac** module callback routine *HAL_DAC_ConvCpltCallbackCh1*. That routine sets a flag in the **dac** module which tells the fillOutputBuffer routine, the next time it is called from the **audio** module, to fill up the last half of the DAC buffer by repeatedly calling nextSample in the **wave** module.

Module Catalog

When the module interaction diagrams are created or revised, attention can be given to the task of creating or updating a **software module catalog**

The Module Catalog component of the detailed design describes each software module to be coded, including its data structures, public functions or methods, and significant private functions. It may also include suggestions for verification testing of the software modules.

The purpose of the catalog is to provide enough information to code the included modules. The appropriate level of detail of the module catalog depends on a variety of factors. If the project is small and the programmer has good access to the designer, the module catalog will probably contain less information than it would contain for a large project.

If class hierarchies, with inheritance and polymorphism, are needed for a particular system, the class diagram defined in UML is a suitable replacement for the module overview diagram, and the UML class catalog can replace the module catalog. When the detailed design is done, the module catalog may be broken into subsets and handed out, with the diagrams, to different members of the software team. The diagrams will help them develop their own mental map of the system. The catalog pages define the scope of their contribution to the overall effort.

Digital Trombone Modules

Module name: main

Initializes the digital Trombone code and enters the main loop which monitors the audio volume produced by blowing into the pressure sensor, monitors the pitch produced by touching the soft pot strip, and produces and manages sounds resulting from the pitch and volume modules.

Source files: main.c main.h

Data elements:

Name	Type	Purpose
none	none	none

Public and significant private functions or methods:

Name	Returns	Arguments	Purpose
SystemClockInit	none	none	Initialize the system clock
main	none	none	Initializes the hardware access layer, configures the clock, and initializes the audio module

Suggestions for verification:

1 Set up main loop to toggle LED3 once per second. Verify that works as expected.

Module name: audio

The **audio** module initializes all the other modules in the system, and enters an endless loop which monitors the **switch**, **blow**, and **slide** modules, calls the fillOutputBuffer function of the **dac** module. If the return code is non-zero, it calls the updateAudioParams function.

Source files: audio.c, audio.h

Data elements:

Name	Type	Purpose
none	none	none

Public and significant private functions or methods:

Name	Returns	Arguments	Purpose
audioInit	none	none	Initialize LED3 which is toggled every time a buffer is output. Initialize every other module in the program.
manageAudio	none	none	call the switch, slide, and blow state machines in their respective modules. call the fillOutputBuffer function of the dac module.
updateAudioParams	none	none	update current audio envelope, frequency, and waveform, and update the flag that tells whether the user is blowing into the pressure sensor.

Suggestions for verification:

- 1 Verify that the manageAudio function is called repeatedly from the main loop, and calls the switch, slide, and blow state machines each time it is called.
- 2 Verify that it calls the fillOutputBuffer method of the **dac** module each time it is called.
- 3 Verify that if fillOutputBuffer returns 1, manageAudio calls the **audio** module updateAudioParams method.
- 4 Verify that when it is called, the updateAudioParams method calls the appropriate methods in other modules: slideSetFrequency in **slide** module, blowOnOff in **blow** module, checkSwitch in **switch** module, and manageNote in the **note** module.

Module name: blow

The user blows into a tube which ends on the pressure transducer. The **blow** module uses an i2c peripheral to read the transducer output. The 14 bit unsigned integer output is proportional to the

pressure produced by blowing.

Source files: blow.c, blow.h

Data elements:

Name	Type	Purpose
blowPressureRaw	unsigned short	14 bit value returned pressure sensor
blowPressureMin	unsigned short	14 bit value read during blowInit, before user blows into tube
blowPressureRange	unsigned short	value produced by subtracting blowPressureMin from maximum possible pressure reading (0x3FFF)
blowPressureRelative	unsigned short	difference between blowPressureRaw and blowPressureMin

Public and significant private functions or methods:

Name	Returns	Arguments	Purpose
blowInit	none	none	Initializes i2c interface, reads blowPressureMin, sets blowPressureRange, and initializes the monitorBlow state machine
monitorBlow	none	none	executes the monitorBlow state machine which reads the latest raw pressure, and sets blowPressureRelative
blowOnOff	none	none	maintains the BLOWING/NO_BLOWING distinction by comparing relative pressure to BLOWING and NOT_BLOWING thresholds
blowPressure	none	none	returns the value of blowPressureRelative
blowPressureMax	none	none	returns blowPressureMax

Suggestions for verification:

- 1 Determine that each time the program powers up, that the value of blowPressureMin is not zero.
- 2 Verify that blowing lightly into the pressure tube produces a blowPressureRelative number greater than the BLOWING_THRESHOLD.
- 3 Verify that barely blowing or not blowing into the pressure tube can produce a blowPressureRelative number less than the NOT_BLOWING_THRESHOLD.
- 4 Suck on the pressure tube and verify that blowPressureRelative goes to zero.
- 5 Verify with the oscilloscope that the audio envelope is effectively zero when not blowing into the pressure tube.

Module name: slide

The hardware applies a 3.3 volt signal between the high and low pins of the slide potentiometer. The **slide** module uses an ADC peripheral to read the voltage between the low and middle pins. Then it uses the ADC count in an algorithm that chooses a corresponding midi note frequency in the normal range of a tenor trombone.

Source files: slide.c, slide.h

Data elements:

Name	Type	Purpose
slidePos	unsigned short	12 bit ADC count returned by reading slide position-dependent voltage

Public and significant private functions or methods:

Name	Returns	Arguments	Purpose
slideInit	none	none	Setup A/D for reading the voltage between ground and position-dependent voltage
slideSetFrequency	none	none	Set the wave frequency to frequency corresponding to slide position
monitorSlide	none	none	state machine reads slide position, and set wave frequency proportionally between E2 and A5

Suggestions for verification:

1 Move the slide from low end to high end and verify that the pitch changes smoothly from E2 to A5, producing no crack or popping sounds using any available waveform.

Module name: switch

This module watches the user switch input, and debounces and saves the latest value. When the button is pressed, it sets the pressed flag which is not cleared until function checkSwitch is called.

Source files: switch.c switch.h

Data elements:

Name	Type	Purpose
switchPressed	0 or 1	0 $\Rightarrow$ not pressed since last check 1 $\Rightarrow$ pressed since last check

Public or significant private functions or methods:

Name	Return Type	Arguments	Purpose
switchInit	none	none	Setup port to read switch. Initialize monitorSwitch state machine
monitorSwitch	none	none	Debounce and latch switch presses.
checkSwitch	1 or 0	Read and clear the pressed variable.	

Suggestions for verification:

- 1 Blow into the mouthpiece to produce a sound. Verify with that the waveform changes from saw to square to triangle to sine and back to saw when the switch is pressed repeatedly.
- 2 Blow into the mouthpiece to produce a sound. Verify that no crack or popping sounds are produced when randomly timed button presses are made.

####Module name: note

This module follows the progress of note production from NO_NOTE to BLOWING_NOTE to RELEASING_NOTE, sets the note envelope when the user is blowing, reads the note envelope when the note is releasing, and ends the note when the envelope falls below the NOTE_OFF_THRESHOLD.

Source files: note.c note.h

Data elements:

Name	Type	Purpose
none	none	none

Public and significant private functions or methods:

Name	Return Type	Arguments	Purpose
noteInit	none	none	Initializes the envelope by calling waveSetEnvelope(0.0,0). Then sets the noteManage state to NO_NOTE
manageNote	none	none	operates the manageNote stateMachine which computes and sets the wave envelope to start the next output buffer while in NO_NOTE and BLOWING_NOTE states, reads the wave envelope in the RELEASING_NOTE state, and goes back to the NO_NOTE state when the wave-generated envelope falls below the NOTE_OFF_THRESHOLD.

- algorithms:*

With an actual trombone, the envelope of each note is completely controlled by the diaphragm, lungs, tongue, and lips of the person playing the instrument. The digital trombone is similar, except that notes have a synthetic release state in which the note envelope falls exponentially to zero. Hopefully this mimics the decay of a trombone note when the user stops blowing.

In the `BLOWING_NOTE` state the final envelope for the next output buffer is set to the floating point ratio between the current `blowPressure` and the `maxBlowPressure`. The `wave` module then interpolates the envelope values between the end of the last buffer output and the end of the next output buffer.

In the `RELEASING_NOTE` state, the current value of the envelope is obtained by querying the `wave` module which computes it for every sample as a decimal fraction times the envelope of the previous sample. When the `wave` module returns a value less than the `NOTE_OFF_THRESHOLD`, the `RELEASING_NOTE` state changes the `manageNote` state to `NOTE_OFF`.

Suggestions for verification:

- 1 Use an oscilloscope on the DAC output pin to verify that blowing into the pressure tube creates an audio envelope whose profile varies with the intensity of blowing, and which falls exponentially to zero when the blowing stops.
- 2 Use an oscilloscope to verify that when the player blows hard, that the DAC output envelope flattens at an amplitude of roughly 1 volt.
- 3 Examine oscilloscope output for any regular variations in output voltage not attributable to user blowing into the pressure tube. In particular check for any flutter at roughly 24 or 42 Hz, reflecting the buffer refill frequency. Also look for discontinuities in envelope and waveform when not pressing the switch. Verify that no such anomalies exist

Module name: `midi`

Computes the frequencies and (hopefully) corresponding slide pot counts for all the notes in the tenor trombone range of E2 to A5. Uses the high and low frequency values to compute, after each output buffer refill, a frequency value proportional to the slide count.

Source files: `midi.c` `midi.h`

Data elements:

Name	Type	Purpose
none	none	none

Public and significant private functions or methods:

Name	Returns	Arguments	Purpose
SystemClockInit	none	none	Initialize the system clock
midiInit	none	none	Computes frequency and approximate slide pot counts for every note between E2 and A5
midiFreq	ushort	ushort	Computes and returns a frequency value corresponding to the slide count supplied

Suggestions for verification:

- 1 Position the slide to the lowest note position and verify that the note matches E2 on the piano or other instrument.
- 2 Position the slide to the highest note position and verify that the note matches A5 on the piano or other instrument.

Module name: wave

The wave object has templates for each waveform (just four in this example). It maintains the current waveform phase, frequency and envelope, and returns the next sample value on request. The phase is incremented by an amount proportional to the frequency divided by the sample rate each time a value is returned.

Source files: wave.c wave.h

Data elements:

Name	Type	Purpose
wavetype	int	Tells which waveform template to use
waveform	int [][]	Waveform templates. One array for each waveform
phase	unsigned int	Accumulated phase
frequency	unsigned short	current frequency
envlp	float	current envelope

Public and significant private functions or methods:

Name	Return Type	Arguments	Purpose
waveInit	none	none	Computes the waveform templates, and zeros the accumulated phase
waveSetFrequency	none	ushort	Sets the current frequency to the value supplied
waveSetEnvelope	none	float	Sets the current envelope to the value supplied
waveGetEnvelope	float	none	Returns the current envelope

| nextSample | none | none | Increments the current phase, wrapping if necessary, and computes the index into the appropriate waveform template. In the NOTE_RELEASE state of the noteManage state machine, it multiplies the envlp value by a decimal fraction and computes and returns the next sample value. In the NOTE_BLOWING state of the noteManage state machine, it sets the envelope value to a number interpolated between the end value of the previous buffer, and the end value for the current buffer. Then it computes and returns the next sampleValue.

nextWaveform none none sets the wavetype to the index of the next
waveform template, wrapping back to the first one
if necessary.

Diagrams, algorithms, etc:

The waveform buffer contains DMA_BUFSIZE/2 samples. One cycle of the waveform is represented in the waveform templates, which have DMA_BUFSIZE/2 entries. The phase variable , accum_phi, varies from 0 to 99,999, representing phase angles from 0 to 2*Pi radians. The change in phase from one sample to the next is:

$$\text{delta_phase} = 100000 * \text{frequency} / \text{SAMPLE_RATE}$$

The next value of phase is:

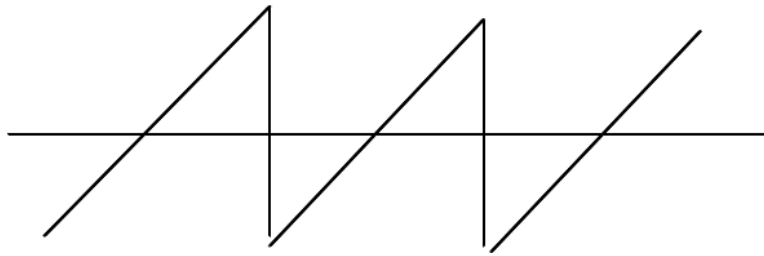
$$\text{phase} = (\text{phase} + \text{delta_phase}) \% 100000$$

The index of the next wave template value to return from nextSample() is computed according to the formula:

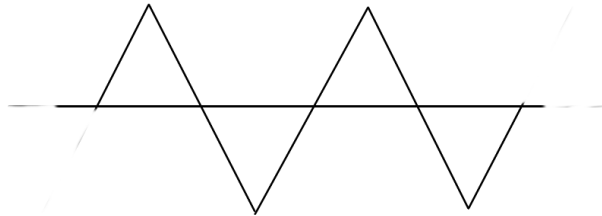
$$\text{Index} = (\text{DMA_BUFSIZE} / 2 - 1) * \text{phase} / 100000$$

Suggestions for verification:

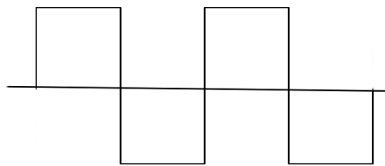
1 Hook an oscilloscope up to the audio output jack. Verify that immediately after powerup the output is a sawtooth waveform, as shown below:



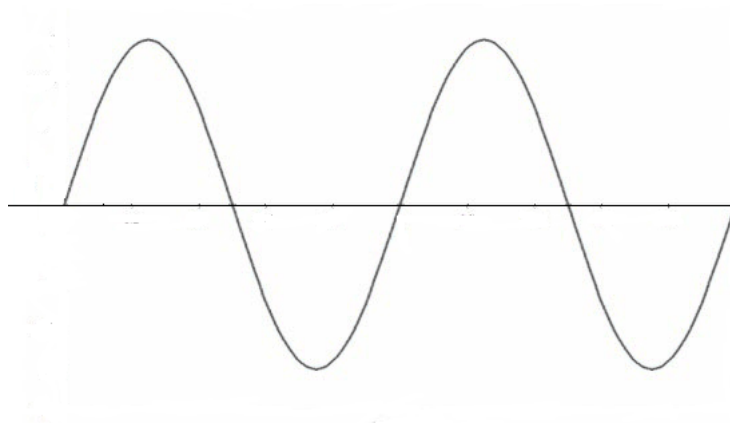
2 Verify with the oscilloscope that the sawtooth waveform changes to a triangle waveform the first time the user button is pressed.



3 Verify with the oscilloscope that the triangle waveform changes to a square waveform the next time the user button is pressed.



4 Verify with the oscilloscope that the square waveform changes to a sine waveform the next time the user button is pressed.



5 Verify with the oscilloscope that the sine waveform changes back to a sawtooth waveform the next time the user button is pressed.



Module name: dac

Computes the frequencies and (hopefully) corresponding slide pot counts for all the notes in the tenor trombone range of E2 to A5. Uses the high and low frequency values to compute after each output buffer refill a frequency value proportional to the slide count.

Source files: dac.c dac.h

Data elements:

Name	Type	Purpose
dacOutBuffer	unsigned int[]	circular buffer for the DMA used by the HAL DAC module
firstHalfFill	unsigned byte	flag indicating it is time to refill the first half of the output buffer
lastHalfFill	unsigned byte	flag indicating it is time to refill the last half of the output buffer

Public and significant private functions or methods:

Name	Returns	Arguments	Purpose
dacInit	none	none	Uses HAL function HAL_TIM_MspInit to initialize TIM6, the timer that controls the rate of outputting samples. Uses HAL function HAL_DAC_MspInit to configure the Timer, DMA, and DAC channel used to convert digital output to analog. Initializes the LED3 port bit to support toggling it in fillOutputBuffer.
startDac	none	none	Sets the firstHalfFill flag and calls DAC_Ch1_Config() to begin operation of the chosen DAC channel at the sample rate chosen (22050 Hz).

Name	Returns	Arguments	Purpose
fillOutputBuffer	unsigned char	none	Checks the firstHalfFill and lastHalfFill flags. If one of them is set, toggles LED3 and then fills the indicated half of the dacOutBuffer by calling the wave module method nextSample WAVE_SAMPLES times and saving the returned value into the next location in the dacOutBuffer
seeNote1	none	DAC_HandleTypeDef *	Overrides weak HAL DAC interrupt half complete callback function
seeNote1	none	DAC_HandleTypeDef *	Overrides weak HAL DAC interrupt half complete callback function

Note1: Actual name of function is HAL_DAC_ConvHalfCpltCallbackCh1

Note2: Actual name of function is HAL_DAC_ConvCpltCallbackCh1

Suggestions for verification:

- 1 Verify that when the program is running, LED3 toggles at the a rate of $22050 / \text{WAVE_SAMPLES}$ = approx 43 Hz.
- 2 Listen to the sound produced for a 43 Hz flutter. Check with a spectrum analyzer to verify that any peak at 43 Hz is much smaller than any peak for the frequency selected or its harmonics.

##Design In Safety

Embedded systems are frequently used in circumstances where system failure endangers life or property. This is particularly true of military, and medical systems. It is also a consideration in automotive, laboratory, industrial, and consumer systems.

In the section on Analysis, the Hazard Analysis was mentioned as a tool to investigate the hazards presented by the new system. In the design portion of the project, other tools may be used to promote safety in the end product. These tools are the Failure Mode Effects and Criticality Analysis and the Fault Tree Analysis.

Failure Mode Effects Analysis (FMEA) This is a formal technique for examine the effects of hardware or software faults in components of the system, and assessing their potential for creating safety hazards. This technique is best applied during the design portion of the project, when some knowledge has been developed of the components, both real and virtual of the system. For more information, search the web for “Failure Mode Effects Analysis”.

Fault Tree Analysis The FMEA looks at safety from a bottom-up view. If this component breaks, what hazards might that present. The fault tree looks at safety from the top-down view. Starting with the list of potential hazards, you ask: “What has to go wrong for this to happen?”. Then you design in software or hardware changes to prevent that fault from happening. For more information, search the web for “Fault Tree Analysis”.

These tools will help you decide where the greatest risks are in the system, and how you might effectively counter those risks. More information on the subject of software safety is available by searching the web with the keywords “Software Safety”.

##Design Review

Design review is a long tradition in the engineering profession. It carries a bit of emotional charge because engineers are putting their creative work up for critical review by their peers. This can be a productive activity, that points up shortcomings or suggests improvements in the design. It can also be a useless exercise in ego stroking, or defensive argument.

On balance, the design review is worth doing. It often results in an improved product.

The best way to approach the design review is to keep it as low key as possible. The designer should invite those people he thinks most likely to contribute. The review should take place in a series of small meetings over a period of days, or even a couple of weeks. Only those persons who are in a position to contribute to the technical process should attend.

All others, especially upper management and financial stakeholders, should stay away, if that is possible. They are apt to misunderstand engineering discussions.

Never make a design review a critical milestone related to payment of development funds. That will cause the review to be rushed, and carried out in a possibly adversarial manner.

Make sure the reviewers have read the documents before coming to the meetings. If they have not, cancel the meeting.

Take notes on any action items that result from the design review, and put those notes in the project archive. Depending upon the kind of process used by your organization, the notes may be needed in a project audit.

Chapter 11: Code

If the analysis and design activities are done thoroughly, writing the code is an easy task. Nevertheless, preparation for coding sometimes plunges the development team into minor conflicts.

No two software engineers write code in exactly the same way. Each engineer has his own preferences and cognitive style. The three most likely conflicts are: Choice of Language, Choice of Operating System, and Coding Standards.

Choice of Language

Most embedded systems are written in 'C', but there are alternatives. C++ is thought by some to provide better tools for translating object-oriented design concepts into source modules. Java and C# are also desirable languages, partly because of the marketing efforts of Sun Microsystems and Microsoft; but also due to their large feature sets, and useful libraries.

Object-orientation is far more important in the design phase than when writing code. Once the code is compiled, there is no important difference between code written in C++ and code written in C. The need for writing strictly typesafe code in C++ often complicates implementations involving function tables and registration-callback patterns.

Some of the desirable features of C++, such as exception processing and virtual functions, come with increased code size, or performance limitation. Java and C# have issues with interpreters and garbage collection that might preclude their use when interrupt latency is a consideration.

The author's view is that C++ is better suited to large systems having many objects of each class, or a large, intricately factored code base, than it is to small or medium sized embedded systems. If you are able to use 'C' in your embedded project, you will seldom go wrong in doing so.

Nevertheless, it may be necessary to use a C++ compiler, or Java, or even C# in order to obtain access to desirable libraries or features. If that happens, take advantage of the choice to learn all you can about implementing useful design patterns in those languages. It is sometimes a challenge, and almost always interesting.

A choice of C++ need not incur performance disadvantage, provided that one does not use the exception processing. You can write a perfectly good C program and convert it to C++ by simply changing the source module suffixes from .c to .cpp. That may be a good approach when the decision is made to develop in C++ to gain access to libraries, or just because somebody said so.

Choice of Hardware Access Layer

Not so long ago, one of the most tedious tasks in embedded system programming was the meticulous cataloging of every hardware register in the CPU and all of its peripherals, as well as making #defines for every value that you might want to assign to a register you have chosen to use.

After a while, the processor vendors took on the task of doing this for you, supplying a suite of header files with more or less information about how the registers should be used.

As processors shipped with more powerful on-chip peripherals, it became more difficult to perform application functions with those peripherals.

The chip vendors such as TI and ST sought to solve that problem by providing Hardware Access Layers (HALs) for their processors. These layers provided a lot of functions that previously developers had to write themselves. Then ARM defined a flexible HAL, called MBED, which changed little from processor to processor.

The Arduino HAL provided the simplest possible hardware access for users of Arduino boards. It was so easy to use that some chip vendors started building Arduino compatibility into their development boards.

The HALs have indeed helped to reduce development time, provided the developers invest the time needed to understand the structure and best ways to use the HALs. That learning curve is a fraction of the learning curve needed to write code that directly accesses the processor peripheral registers.

The success of HALs is most likely due to the fact that most software developers find it easier to access peripherals through a software system, than to learn how the hardware works.

Conversely, some electrical engineers still prefer programming the hardware registers directly, because that is more familiar to them than using some complicated software access layer.

ST's HAL for the STM32L4 series of processors helped to shorten development time for the Trombone example.

An alternative would have been to use the MBED or Arduino HALs for the project, as the chosen development board is compatible with both. For this project, the ST HAL provides more help with ST processor peripherals than Arduino. The general purpose MBED HAL can be harder to adapt to a processor than one furnished by the chip vendor.

##Choice of Operating System

Many embedded applications require multiple parallel processes. There might, for example, be one process which operates a communications protocol, one which manages a robotic arm, and one process which interacts with a user over a keypad.

The mainframe and PC solution for such situations is a multi-tasking operating system. Each process gets its own logical task, implemented by function calls from a for or while loop. The communication between tasks is implemented with semaphores, critical sections, pipes, or even network protocols.

The use of a PC operating system has more to recommend it than just handling multi-tasking. It allows the developer to use common PC components for the user interface, and network communication. This can save a bundle in engineering labor re-inventing common display and networking components, both hardware and software.

The downside of the PC OS approach arises when the developer needs to write a driver for custom hardware to allow its use with the PC. The cost of implementing drivers for PC operating systems can be a lot more than one might expect.

When using a PC operating system, there is another problem as well. The user interface component will often be written in an interpreted language that is different from the language needed for any custom robotic components of the system. This forces the developer to connect the user interface to the hardware drivers through a command language and comm protocol.

Configuration control of the Linux operating system is a thorny issue, since it is continually undergoing changes from different people and organizations separated by geographical and cultural differences.

Using the Windows operating system exposes one to Microsoft's tendency to craft technology features that coerce increased use of Microsoft products in customer environments.

A special class of operating systems called real-time operating systems (RTOSs) mitigate the code size, and interrupt-latency issues common with PC operating systems. They may also require considerable use of scarce system resources, such as processor time and memory.

Sometimes even the RTOS's interrupt latency is too much for some board designs. The RTOS also forces the use of elaborate thread-safe interprocess communication tools, which might not otherwise be necessary.

An alternative to the use of a PC operating system or RTOS is the use a main loop which invokes separate state machines for each process. This technique requires the individual processes to be divided into small chunks, each of which is implemented as a separate state of the processes's state machine.

Each time a state machine is called by the main loop, only one of its states executes. When that chunk of the state machine finishes, it transfers control to another chunk or "state", which is called the next time through the main loop.

The state machine approach substantially reduces overall code size compared to an RTOS implementation. It also simplifies interprocess communication, since one process state is never interrupted by another process state. And it reduces interrupt latency, since there is no operating system code which must be protected by disabling interrupts. There remain, of course, thread safety issues related to communication with interrupt routines.

Coding Standards

A British philosopher once observed that there are really only two kinds of people in the world: the simple-minded, and the muddle-headed. On software development teams, that dichotomy often

manifests as those who are sticklers for the appearance of source code, and those who prefer broad latitude for personal expression.

The sticklers want the look of source code to be tightly controlled. The manner of nesting braces must conform rigidly to a preferred scheme. Capitalization or non-capitalization of variable and function names must be subjected to rigorous collections of rules. All flow of control statements must use braces, whether they need them or not.

The broad brush ones don't appreciate nitpicking of their artistic deployment of coding symbols. They usually accept coding standards that affect the object code, but purely stylistic issues are seen as invasion of their personal prerogative.

This conflict is probably a reflection of different cognitive styles. Persons in the two groups simply use their brains in different ways. Fortunately, there are tools to resolve this issue without creating conflict.

If you are a member of a team that has chosen a coding standard that doesn't fit your worldview, purchase an editor that supports multiple styles of pretty-printing. There are several such editors, and most support the generally favored coding styles of sticklers. Then you can write your code any way you like, but you use the editor to reformat it before checking it into the project repository.

Coding the Trombone Project

In our digital Trombone code, C is the chosen language and the ST HAL provides the best peripheral access.

Only one task is used in the digital Trombone code, along with a SysClock interrupt, a sample-rate timer interrupt, and the DAC interrupt, so no operating system was needed.

Complete source code for the Trombone code, is available as a no-cost extra when this how-to-develop eBook is purchased.

Chapter 12: Debug

Debugging presents a continuing series of riddles for the developer. Each riddle is a consuming mystery, right up to the point when it is discovered to result from an obvious mistake. Coding, Debugging, and Integrating often merge into a single confusing process. Here we arbitrarily define debugging to be the work done on individual modules to make sure they perform according to their published interfaces (public function and method specifications). This work is usually done by the person writing the code, with a little help from his friends, if any.

Bug Categories

Most bugs fit into one or more categories. It helps to bear these categories in mind during debug and integration. They serve as hypotheses to account for the deep mysteries of your bugs. Below is a list of common embedded system bug categories, some of which occurred in the digital Trombone project. The list is not exhaustive. In fact, it covers only a fraction of the problems you will encounter. You can find other bug lists on the internet, or in books devoted to software testing.

One-off errors: These errors usually consists of starting or stopping a loop one iteration too soon or one iteration too late. These can easily result in memory corruption or buffer overflows (see below).

Buffer overflow errors: Continuing to copy data beyond the end of buffer causes memory corruption errors (see below).

Type casting errors: Casting one type of variable as another type is always a suspicious activity. That's why Lint pays such close attention to improper type casting. It is a particularly good candidate if you have a bug in an arithmetic algorithm.

Careless syntax errors: We all make this kind of mistake. It can be hard to catch without the help of other team members.

Poor thread safety: Accessing the same variable from two separate threads should be done only when exclusive control of the variable can be guaranteed.

Bad pointer errors: These lead to memory corruption errors and a variety of other conditions arising from reading unexpected values.

Variable name errors: Make sure that your variables have the names you think they do. If you misspelled a variable name, and it happened to correspond to the name of another variable in the system, you've got problems.

Unhandled exceptions: Divide by zero errors are common glitches of this type. Unnoticed overflow is another. Floating point routines are particularly susceptible to this problem. Data Aborts are also common, arising from accessing data at an address incompatible with instruction requirements.

Inadequate design: This usually results from failure to consider all possibilities in an algorithm, a data structure, or an object interaction. That, in turn, often happens when there is a rush to write code, at the expense of design effort.

Pipeline errors: In pipelined processors, instructions are not always executed in the order they are read by the processor, particularly when interrupts are processed. I once had a pipeline-related error that prevented interrupts from being disabled, even though the disable instruction itself was clearly being fetched.

Stack overflow: An easy bug to fix, this fault can produce a bewildering variety of symptoms. This ought to be an early suspect in memory corruption, and weird execution sequence bugs. Just add more stack and see if that fixes or delays the problem.

Memory allocation errors: Freeing unallocated memory, freeing the same memory twice, and overwriting memory accounting fields have long troubled systems with dynamic memory allocation. Even if you don't have a heap, you may still have allocation errors in buffer managers, pipe managers, and linked list node arrays.

Misunderstanding the hardware: This includes a variety of problems, such as setting up chip select signals incorrectly, setting the clock to the wrong speed, and failing to specify the correct number of wait states for blocks of memory or I/O space. On processors supplied with powerful hardware peripherals such as flexible serial ports, DMA's, and timer modules, it is easy to botch the setup of a hardware configuration. Some processors have different modes of arithmetic or addressing operations that must be carefully set and monitored. Careless I/O port assignment Getting the port mask wrong for a signal means you are not looking at the signal you thought you were seeing. Alternatively, you are not driving the signal you thought you were driving.

Operator precedence errors: Parentheses take care of most of these errors, though not all. In the example below, a wicked memory corruption bug arose from the sequence:

```
if(k<N) {  
  x[ k ] = a + x[ k++ ];  
  break;  
} else continue;
```

The problem here is that the post incrementation takes place after the limit check on k, but before the assignment. The result is a variable one word past the end of x[] is modified by the assignment.

Variable scope errors: These can be largely avoided by avoiding the same names in automatic, static, and global variables. It is also handy to use an object prefix for all the variables in an object, or to make all such variables members of a structure that contains all the object variables.

Link editing errors: It is a good idea to verify that memory actually exists at every location where a variable is placed by the linker, and that it is the type of memory that you were expecting for that

variable.

Variable alignment errors: Some processing operations only work when variables begin on a long word boundary. Some compilers align variables which are structure members on byte, word, or double word boundaries depending upon compiler switch settings. Some DSP buffers must be aligned on boundaries having a number of trailing binary zeros equal to the that in the next power of 2 greater than or equal to the buffer size. Failure to comply with all such constraints creates memory corruption bugs.

Algorithmic errors: Are you sure you implemented that algorithm correctly?

Memory corruption errors: Memory corruption can manifest in a bewildering variety of ways. You may be overwriting any pointer or variable or piece of RAM-resident code in the system. The fault may not become apparent for many seconds, after which it is too late to trace it to its source. Deduction and logic analyzers are good tools for attacking memory corruption errors. Checksumming sections of code or data can also help.

Timing errors: You need a good multi-channel oscilloscope or a logic analyzer to investigate timing bugs. Find or create signals which show all of the timing relationships in the problem area. Then look at those signals on the scope to make sure things happen at the rate, and in the order expected. Oftentimes, just putting the problem on the scope is enough to find the cause.

Initialization errors: Every compiler and linker outputs memory reservations for something called a BSS area. That is the Binary Storage Section. It contains the locations for all of those variables which are not constants or statically initialized. Usually the C startup code will zero the BSS area, but in some situations it may not. The upshot is this: always specifically initialize every variable used in a program. Initialize that variable both statically and dynamically. The static initialization should be good enough unless your system has a warm-start capability, which may restart the main program without going through the complete boot-up process. Then you also need the dynamic initialization.

Warm start errors: Warm start refers to a situation in which the code is restarted without powering down the system. Since the system is not powered down, RAM memory contents are not destroyed. Sometimes the compiler initialization code is not run, or is not run in the same way. Warm starting is frequently parameterized with an error code left over from the previous run of the system, or some other remnant of a previous run. Every different way of warm starting the system offers a completely new set of initialization errors to explore. Each power-up and parameterized warm start must be understood in complete detail as to sequence of activities, and expected variable values. If at all possible, avoid warm starting the system all together.

Circuit board errors: Circuit designers make mistakes too. When they do, you may think you are reading a serial clock signal when you are actually looking at the ground plane. The only way to catch these errors is to watch the related signals on the scope to verify that they are working as expected. If they are not, get data sheets for the related chips, and trace the malfunction through the circuit board. Better yet, demonstrate the problem to a hardware engineer, and let him trace the circuit malfunction.

Programmable logic errors: These are like circuit board errors except you cannot trace the signals beyond the pins of the gate array or programmed logic device. For that you need the equations

which were used to program the device. If you find a pin on a programmed logic device that is not producing the desired signal, check first the inputs which are supposed to produce that resulting signal. If they are right, either turn the problem over to a hardware engineer, or consider changing careers and becoming a hardware engineer.

Compiler errors: In thirty years of embedded systems work, I have discovered an average of one compiler bug per medium sized system. That's not to say that there weren't other compiler bugs in each system. I just didn't catch them. For any given bug, there is a small but finite chance it results from the compiler not behaving as expected. Thus, it is always a good idea to look at how the compiler has translated suspect code, to see if this might be that single compiler bug for the project. Don't do this first, though, or even twenty first. This should be the first of the so-called desperation checks.

Library errors: Modern processors are often accompanied by sample library code which exercises most of the features. Evaluation boards also come with more extensive libraries which implement common things such as USB drivers, mass storage drivers, and internet protocols. More often than not, you will find hidden bugs in one or two of the library routines you use.

Spurious interrupt errors: Sometimes even hardware designers forget to do things like tie signals to power or ground. If those signals happen to be hooked up to processor interrupt pins, you can get some pretty mystifying failure modes. Catch these bugs early by writing a spurious interrupt routine that is reached from every interrupt the processor accepts, that is not used for a system interrupt. Make sure that the spurious interrupt routine makes it perfectly clear what is wrong.

Power supply noise: Avoid if possible, working on sensitive electronic instruments which use on-board switching power supplies. If you cannot avoid this situation, be on the lookout for peculiar spikes in signals read by the software. Nothing messes up a simple signal processing algorithm faster than sporadic spiking on its input signal lines.

##Advice for Debuggers

For an embedded software developer, speculating on hardware causes for bugs is evidence of desperation. It may very well be a hardware bug. Several of those crop up during the course of a project; but the usual finding is that the software is at fault.

When another developer tells you he thinks your software has a bug in it, respond immediately. He may be solving your problem for you.

When you have looked for the same bug for a long time, your mind gets stuck. You will not find the bug until you alter your mental frame of reference. Do whatever you normally do to accomplish that feat. This can mean leaving the workplace, or engaging in a variety of stress-reduction behaviors, including sleep. Describe your bug to a team member. For 20% of all bugs this results in the immediate solution of the problem.

Never make major changes to source code late in a debugging session. You invariably find yourself restoring the code to its earlier state, if you saved a backup copy.

Fear and debugging are incompatible. If you are afraid for your job, you will not find the bug. Fear is either due to your own feelings of unworthiness, or to a climate of fear in the workplace. If it is

the former, explore those feelings with a therapist. If it is the latter, find another job.

When you are totally stumped, and you tried standing on your head and that didn't help, consider each of the bug categories above for applicability. Get a book that contains common causes of bugs, such as *Software Testing Techniques*, by Beizer. Find other bug category lists on the internet. You will seldom find a difficult bug without first forming a hypothesis as to its cause.

Never blame the development environment, the tools, or another team member for the bug you are tracking. The bug is in the software, not in the environment, not in the tools, and not in your co-workers. Improve your circumstances if you can, but find the bug regardless of your circumstances. Don't let debugging get you down. You could have been one of those people who have to work for a living.

Chapter 13: Integrate

During integration, the hardware and virtual objects created by the developers are brought together and made to work with one another in accordance with the system design.

With a decent system design, competent work by the developers, adequate debugging tools, and the absence of schedule pressure, integration can be an exciting and rewarding experience. The fewer of the above requirements are met, the more trouble you will have during integration.

Integration is an extension of debugging. You are still looking for bugs, but now they are harder to find because you need the help of all of the other team members. There is usually some anxiety about whether the dang thing will work at all. This anxiety is especially common among resource providers, who by this time, have stuck their neck out a mile and a half supplying the money for the project.

In analysis, design, coding, and debugging, the engineer focuses mostly on what can go wrong. Pessimism is a vital part of the mindset necessary to cope with physical reality. In the integration phase, the precautions have all been taken. You are now looking for everything to fit together and work. A healthy dose of optimism at this point can help you persist through whatever difficulties arise.

The keys to a successful integration are a good design, a cooperative attitude, and optimism. The better the system design, the fewer problems will crop up in integration. The better each team member's attitude, the more likely the team is to pull together.

Advice for Integrators

Integrate early and integrate often. The earlier in the project you can build an end-to-end version of the software (even if most parts are stubbed out), the less confusion you will encounter when everybody is adding code under pressure of deadlines.

Build an early skeleton of the system. Add to it regularly as code becomes available. If target hardware is not available, use the closest eval board you can find. Simulate missing objects, hardware or software, in whatever way is feasible.

Assign one team member the role of point-man for integration. This person will make the first attempts at an end-to-end system, and will provide the initial release that is used by other team members in the early stages of integration.

There will be several configurations of the software under development and test at the same time during integration. Pick a configuration management tool and stick with it. Develop procedures for defining releases, and for tracking the uniquely versioned source modules in each release.

Always keep a baseline release of the system. This is the release that includes working versions of the most virtual objects, without any special debugging additions or enhancements. This release represents your latest and greatest working system. It is the release you will deliver when it finally contains working modules for every object in the system.

Archive copies of the baseline release as often as once a day. That way the most you can lose is a day's work. The integration point-man is responsible for updating the release archive, and for knowing roughly what is in each archived release. In today's high-turnover environment, it is a good idea to have a backup person who is familiar with building the baseline release.

Public finger-pointing or blaming other team members is counter-productive. If you discover a problem with another team member's contribution, explain privately to that person what you found, and ask for their help in resolving the issue. Do not bring it up in a public meeting unless it affects others besides you and the other team member.

Never assume you are not the cause of someone else's problem. Keep up to date on the problems other team members may be having. Spend some time each day asking yourself how you might be responsible for those problems. Try to help the other team members solve their problems even if you are convinced they are not related to your own work.

When you are stumped on a problem, seek help from other team members. Ask the whole team to listen to your description of a problem, and suggest ways to find its cause. Integration does not happen without communication.

Chapter 14: Verify

Verification is a more or less formal way of checking your work. It is like proofreading a document. Verification catches the mistakes you didn't catch when you were debugging or integrating.

In projects where safety is an issue, verification tends to be quite formal. It might include the creation of overall verification plans, the definition of detailed testing protocols, and publishing of reports of every test, its outcome, and whether retesting is necessary. It always includes detailed testing of any code added to the system to mitigate hazards.

In the digital Trombone project, suggestions for verification were included in the Module Catalog part of the design. This section contains the verification report, which lists the tests performed on each object, and the outcomes of those tests.

Module: main

Test Number: 1

Test Description: Set up main loop to toggle LED3 once per second. Verify that works as expected.

Test Outcome: pass

Retest Necessary: no

Module: audio

Test Number: 1

Test Description: 1 Verify that the manageAudio method is called repeatedly from the main loop, and calls the switch, slide, and blow state machines each time it is called.

Test Outcome verified by code inspection and debug breakpoint

Retest Necessary: No

Test Number: 2

Test Description: Verify that it calls the fillOutputBuffer method of the dac module each time it is called.

Test Outcome verified by code inspection and debug breakpoint

Retest Necessary: No

Test Number: 3

Test Description: Verify that if fillOutputBuffer returns 1, manageAudio calls the audio module updateAudioParams method.

Test Outcome verified by code inspection and debug breakpoint

Retest Necessary: No

Test Number: 4

Test Description: Verify that when it is called, the updateAudioParams method calls the appropriate methods in other modules: slideSetFrequency in slide module, blowOnOff in blow module, checkSwitch in switch module, and manageNote in the note module.

Test Outcome verified by code inspection and debug break point

Retest Necessary: No

Module: blow**Test Number: 1**

Test Description: Verify with a breakpoint that each time the program powers up, that the value of blowPressureMin is not zero.

Test Outcome: verified by observing at least twenty power up trials with a breakpoint in i2cInit for observing blowPressureMin

Retest Necessary: No

Test Number: 2

Test Description: Verify by breakpoint that blowing lightly into the pressure tube produces a blowPressureRelative number greater than the BLOWING_THRESHOLD.

Test Outcome: verified by observing at least twenty light blowing trials with a breakpoint in i2cInit for observing blowPressureRelative

Retest Necessary: No

Test Number: 3

Test Description: Verify that barely or not blowing into the pressure tube can produce a blowPressureRelative number less than the NOT_BLOWING_THRESHOLD.

Test Outcome: verified by observing at least twenty very light or not blowing trials with a breakpoint in i2cInit for observing blowPressureRelative

Retest Necessary: No

Test Number: 4

Test Description: Suck on the pressure tube and verify by breakpoint that blowPressureRelative goes to zero.

Test Outcome: verified by observing blowPressureRelative while sucking on the pressure sensor input tube.

Retest Necessary: No

Test Number: 5

Test Description: Verify with a breakpoint that the audio envelope is effectively zero when not blowing into the pressure tube.

Test Outcome: verified by debug breakpoint

Retest Necessary: No

Module: slide**Test Number: 1**

Test Description: Move the slide from low end to high end and verify that the pitch changes smoothly from E2 to A5, producing no crack or popping sounds using any available waveform.

Test Outcome: Verified by by sliding potentiometer up and down entire travel with each waveform.

Retest Necessary: No

Module: switch

Test Number: 1

Test Description: Blow into the mouthpiece to produce a sound. Verify with that the waveform changes from saw to square to triangle to sine and back to saw when the switch is pressed repeatedly.

Test Outcome: Test passed

Retest Necessary: No

Test Number: 2

Test Description: Blow into the mouthpiece to produce a sound. Verify that no crack or popping sounds are produced when randomly timed button presses are made.

Test Outcome: Test passed

Retest Necessary: No

Module: note

Test Number: 1

Test Description: Hook up an oscilloscope between the DAC output pin and ground. Verify that blowing into the pressure tube creates an audio envelope whose profile varies with the intensity of blowing, and which falls exponentially to zero when the blowing stops.

Test Outcome: Test passed by visual observation of envelope on oscilloscope.

Retest Necessary: No

Test Number: 2

Test Description: Hook up an oscilloscope between the DAC output pin and ground. Hook up an oscilloscope between the DAC output pin and ground. Verify that when the player blows hard, that the DAC output envelope flattens at an amplitude of roughly 1 volt.

Test Outcome: Test passed by visual observation of envelope on oscilloscope.

Retest Necessary: No

Test Number: 3

Test Description: Hook up an oscilloscope between the DAC output pin and ground. Examine oscilloscope output for any regular variations in output voltage not attributable to user blowing into the pressure tube. In particular check for any flutter at roughly 24 or 42 Hz, reflecting the buffer refill frequency. Also look for discontinuities in envelope and waveform when not pressing the switch. Verify that no such anomalies exist

Test Outcome: Test passed by oscilloscope observation at a variety of scan rates and voltage settings, as well as by selecting spectrum display of signal for each waveform.

Retest Necessary: No

Module: midi**Test Number: 1**

Test Description: Position the slide to the lowest note position and verify that the note matches E2 on the piano or other instrument.

Test Outcome Test passed

Retest Necessary: No

Test Number: 2

Test Description: Position the slide to the lowest note position and verify that the note matches A5 on the piano or other instrument.

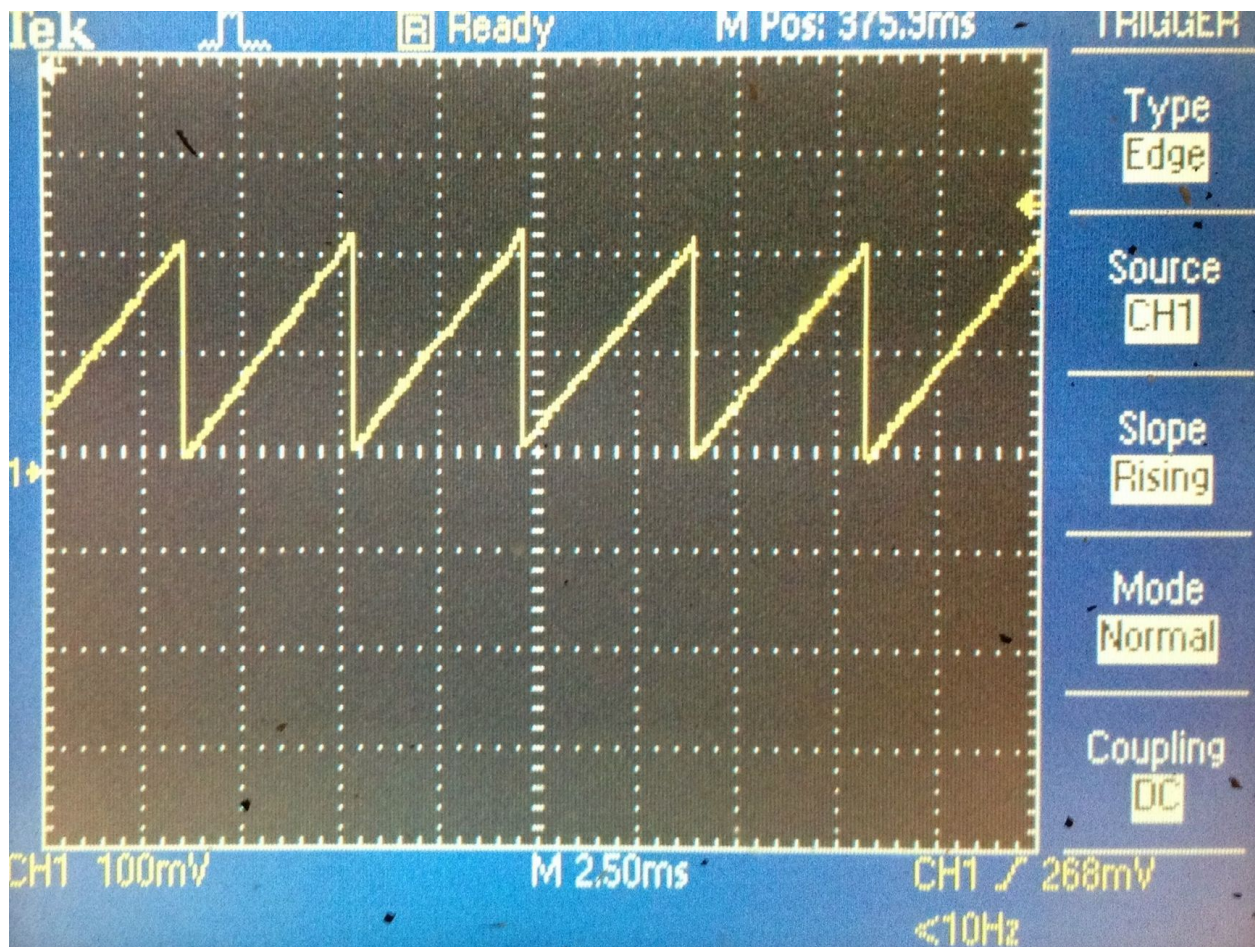
Test Outcome Test passed

Retest Necessary: No

Module: wave**Test Number: 1**

Test Description: Hook up an oscilloscope between the DAC output pin and ground. Verify that immediately after powerup the output is a sawtooth waveform.

Test Outcome

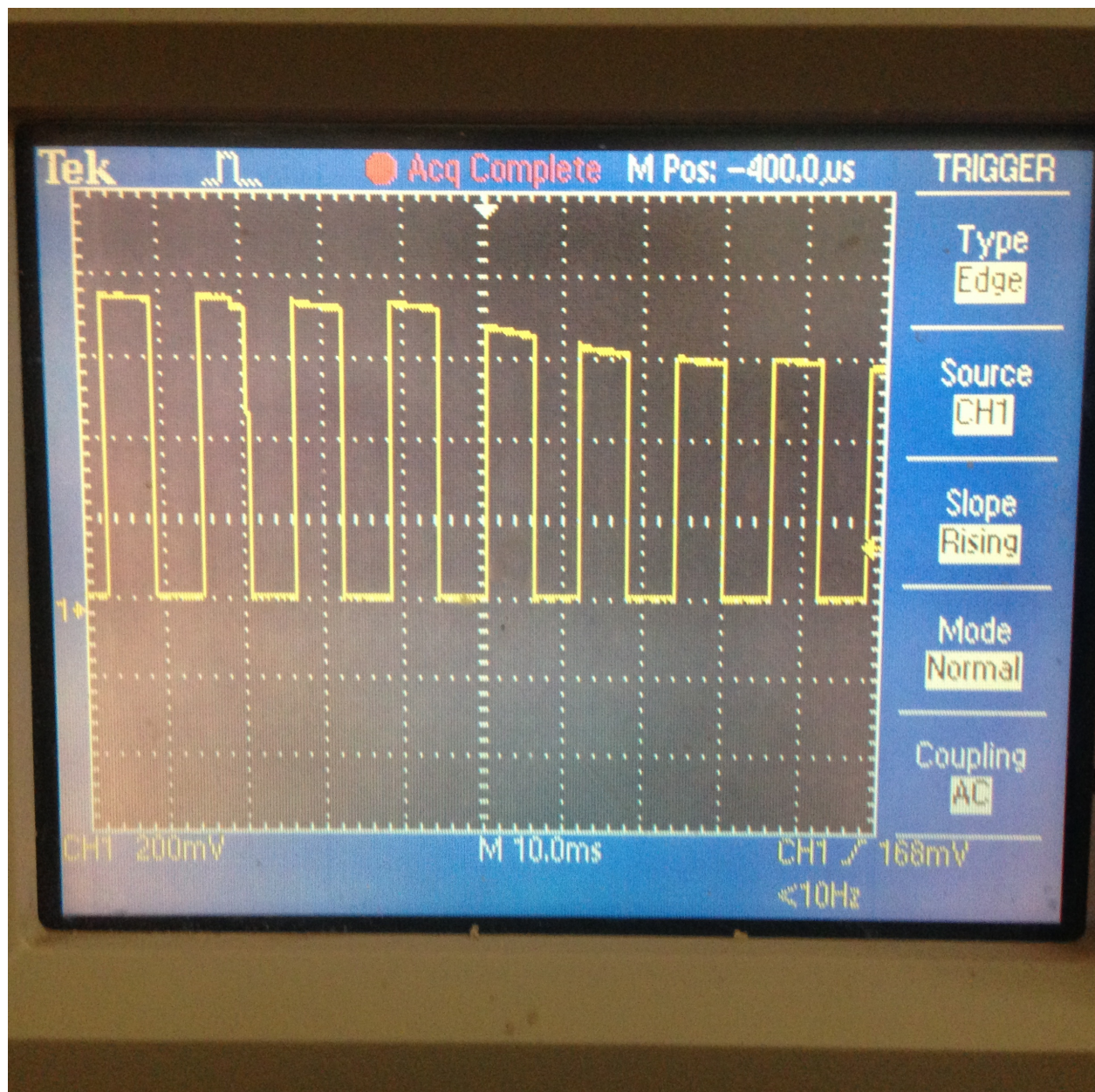


Retest Necessary: No

Test Number: 2

Test Description: Verify with the oscilloscope that the sawtooth waveform changes to a square waveform the first time the user button is pressed.

Test Outcome Test passed

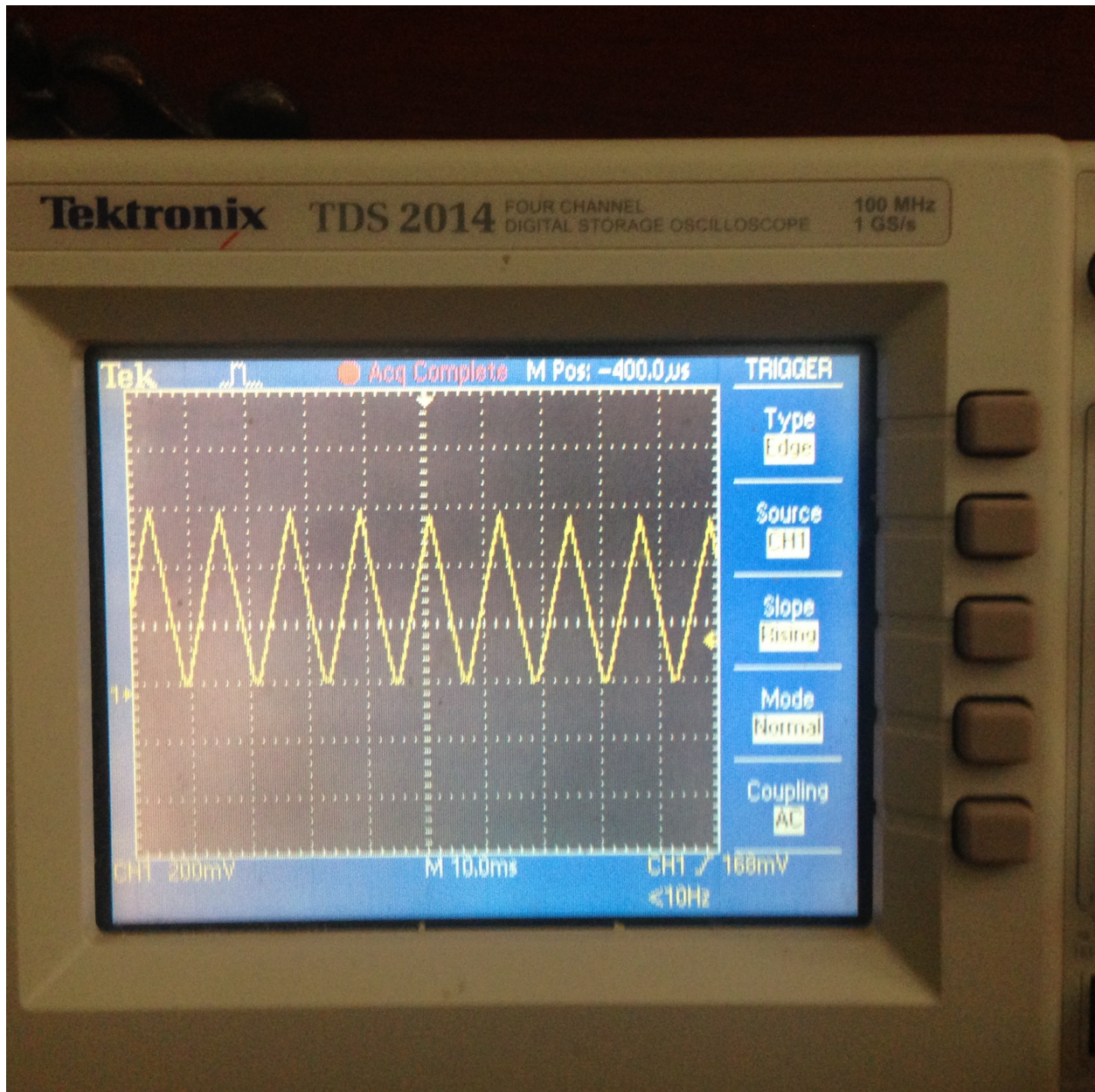


Retest Necessary: No

Test Number: 3

Test Description: Verify with the oscilloscope that the square waveform changes to a triangle waveform the next time the user button is pressed.

Test Outcome Test passed

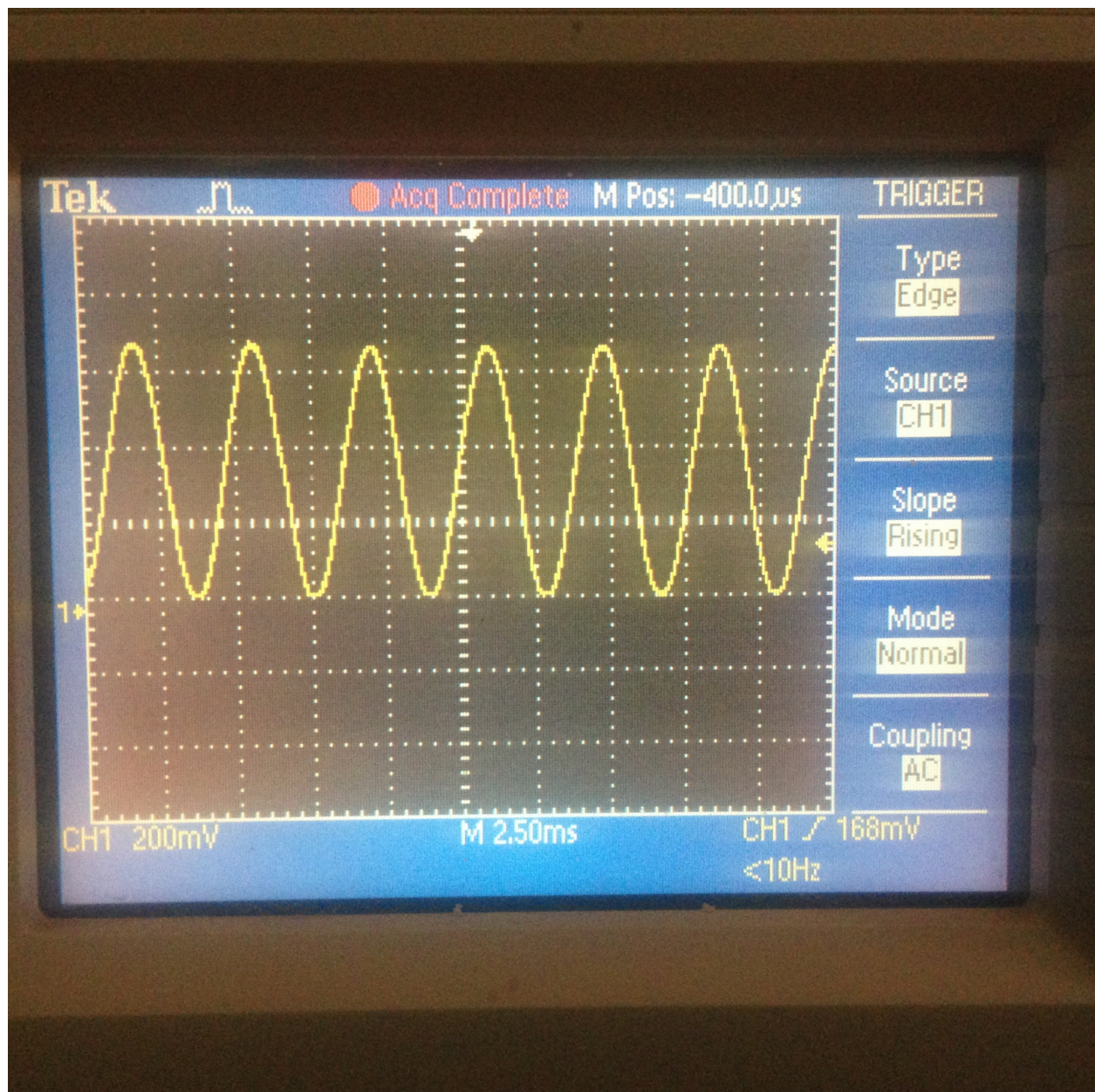


Retest Necessary: No

Test Number: 4

Test Description: Verify with the oscilloscope that the triangle waveform changes to a sine waveform the next time the user button is pressed.

Test Outcome Test passed



Retest Necessary: No

Test Number: 5

Test Description: Verify with the oscilloscope that the sine waveform changes to a saw waveform the next time the user button is pressed.

Test Outcome Test passed

Retest Necessary: No

Module: dac

Test Number: 1

Test Description: Verify that when the program is running, LED3 toggles at the a rate of 22050 / WAVE_SAMPLES = approx 43 Hz.

Test Outcome Test passed

Retest Necessary: No

Test Number: 2

Test Description: Verify that no 43 or 22.5 Hz flutter is heard when blowing into the pressure tube and listening with earphones. Check with a spectrum analyzer to verify that any peak at 43 or 22.5 Hz is much smaller than any peak for the selected frequency or its harmonics.

Test Outcome Test passed

Retest Necessary: No

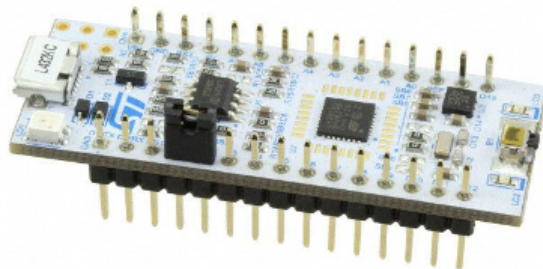
Chapter 15: Validate

Verification is a double check that the system's modules perform and interact as they are designed to interact. Validation is a series of checks to demonstrate that the system as a whole meets its requirements.

The validation report for the Digital Theremin system evaluates each requirement in the formal requirements list against the performance of the system. The formal requirements are listed below in *italics*, with discussion and the manner of validation below each requirement.

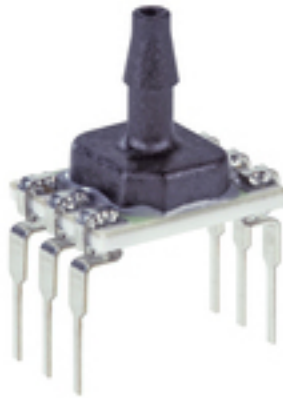
Requirement 1

The digital trombone shall use a NUCLEO-L432KC, mbed-enabled Development Nucleo STM32L4 MCU 32-Bit ARM Cortex-M4 Embedded Evaluation Board. This is an Arduino UNO compatible, Arm Cortex-M4 processor with minimal supporting hardware.



Requirement 2

The Trombone accepts input from a Honeywell Pressure Sensor, part #ABPDANN005PG2A3, desc: DIP, barbed, 5 PSIG.



Requirement 3

Trombone slide shall be a 10 centimeter linear slide potentiometer.



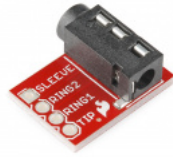
Requirement 4

The User Button will be a two contact, spring-loaded-to-off push button.



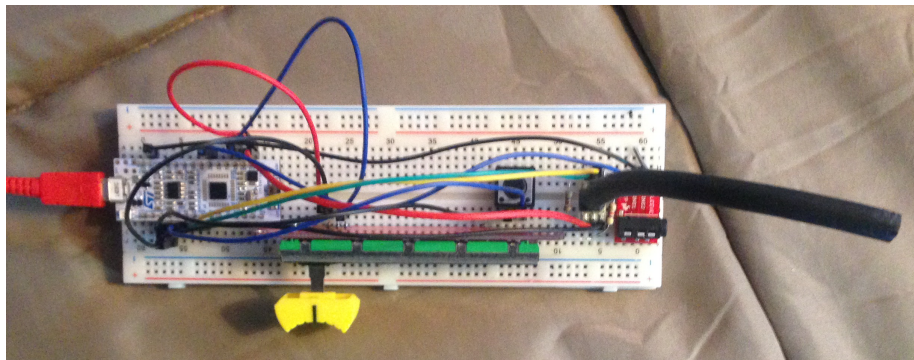
Requirement 5

A headphone jack will be supplied such as the one shown below:



Validation

Below is a photo of the completed hardware configuration:



The parts specified are the ones used, except that a slide potentiometer breakout board with yellow handle was used instead of the one depicted in Requirement 3.

Requirement 6

Within two seconds after power-up, the Trombone enters normal operation with a saw waveform.

Validation

Normal operation is observed to occur within a fraction of a second after power up.

Requirement 7

Every note played consists of a waveform modulated by a pressure envelope.

Validation

Listening with earphones and observing on an oscilloscope demonstrated that this is the case.

Requirement 8

The waveform begins at the frequency determined by the potentiometer position, and a volume determined from the pressure sensed.

Validation

This is demonstrated by varying the potentiometer setting, and then blowing into the pressure tube. The starting frequency does depend upon the pot setting, and the sound volume varies with how hard you blow. Both observations are confirmed by observation of notes on the oscilloscope.

Requirement 9

Individual notes begin when pressure is applied to the pressure sensor, and exponentially decay when it is removed. Between application and removal of pressure, the volume and frequency may vary with pressure and slide position changes.

Validation

Listening to the earphones when you start to blow demonstrates that a note begins when you start blowing into the pressure tube. When you stop blowing, the note is heard to die away. When observing the cessation of blowing on the oscilloscope, the note is seen to die away exponentially.

Requirement 10

The digital trombone will support the same frequency range as a basic tenor trombone (E2 to A5).

Validation

Listen to the note played at the extreme ends of the slide pot, and demonstrates that they are the same as E2 on the piano for the low end, and A5 on the piano for the high end.

Requirement 11

The trombone reacts to a press of the user switch by choosing the next waveform in the circular waveform table for the subsequent note or notes. Waveforms supplied include Saw, Square, Triangle, and Sine.

Validation

Using earphones and an oscilloscope to observe the output, play a note, then press the button and play another note. Both the sound and the oscilloscope trace reveal that the switch changed the waveform. Pressing the button four times in a row demonstrates that all four waveforms are played.

Requirement 12

The waveform may change when no note is currently playing, or in the middle of a note.

Validation

Blowing into the pressure tube, observing the waveform, ending the note, and pressing the button, and blowing again into the pressure tube, and observing the waveform demonstrates that the waveform changes even though no note is being played.

Blowing into the pressure tube, observing the waveform, and continuing to blow while pressing the switch and observing the waveform verifies that the waveform changes with the switch is pressed while playing a note.

Requirement 13

No cracks or pops will be heard except possibly when the waveform is changed in the middle of a note.

Validation

Cracks and pops are not heard in output, except for minor ones sometimes when the waveform is changed while blowing.

APPENDIX A

Cellular Calculator Scripts for Estimating

To obtain an iPhone App, MathWeb, which includes these scripts, look for MathWeb on the iPhone AppStore.

Cell Script for COCOMOII Computation

```
<?xml version="1.0" program="Formulate" encoding="UTF-8"?><topic>
<numfmt>1</numfmt>
<sigdig>4</sigdig>
<fixedec>2</fixedec>
<trigunits>0</trigunits>
<xcell>home</xcell>
<ycell>home</ycell>
<xdivs>1</xdivs>
<ydivs>1</ydivs>
<xmin>0</xmin>
<ymin>0</ymin>
<xmax>1</xmax>
<ymin>1</ymin>
<aplot>1</aplot>
<lwidth>2</lwidth>
<cell><cname>ACAP</cname><text>/*
  * Analyst Capability
```

Rate according to the scale below. Place rating into the value field of this cell.

Extra Low	----
VeryLow	1.46
Low	1.19
Nominal	1.00
High	0.86
VeryHigh	0.71

*/

```
</text><value>1.000</value></cell><cell><cname>Enhanced</cname><text>/*
```

* To enhance your estimate, fill in values for the following factors, using the guidance from the comments in each factor's cell:

RELY DATA CPLX RQUT TIME STOR VIRT TURN ACAP AEXP PCAP VEXP LEXP MODP TOO\

L SCED

*/
out "\nEnhanced Estimate"

\$MM = EnhancedMM*TotEffect

TDEV FSP

out "\n MM=\$MM man months, TDEV=\$TDEV calendar months , FSP=\$FSP developers required" </text> <value>5.596</value> </cell> <cell> <cname>SCED</cname> <text> /*

* Schedule Pressures

Rate according to the scale below. Place rating into the value field of this cell.

Extra Low ----

VeryLow 1.23

Low 1.08

Nominal 1.00

High 1.04

VeryHigh 1.10

ExtraHigh ----

*/

</text> <value>1.100</value> </cell> <cell> <cname>KDSI</cname> <text> /*

* Source statements are all program instructions created by project personnel and processed into machine code by compilers, etc. Delivered source statements excludes nondelivered support software such as test drivers.

In the value field of this cell place the number of thousands of delivered source statements expected.

*/

</text> <value>5.020</value> </cell> <cell> <cname>PCAP</cname> <text> /*

* Programmer Capability

Rate according to the scale below. Place rating into the value field of this cell.

Extra Low ----

VeryLow 1.42

Low 1.19

Nominal 1.00

High 0.86

VeryHigh 0.70

ExtraHigh ----

*/

```

</text><value>1.000</value></cell><cell><cname>RQUT</cname><text>/*
  * Requirement Changes

```

Rate according to the scale below. Place rating into the value field of this cell.

```

Extra Low      ----
VeryLow        ----
Low            0.98
Nominal        1.00
High           1.50
VeryHigh       2.00
ExtraHigh      2.50
*/

```

```

</text><value>1.000</value></cell><cell><cname>MM</cname><text>/* This cell computes\
  the expected number of man months (152 hours) required to produce the number of tho\
  usands of delivered source statements given in KDSI.

```

```

*/
if(mode==1)
    2.4*pow(KDSI,1.05)
else if(mode==2)
    3.0*pow(KDSI,1.12)
else if(mode==3)
    3.6*pow(KDSI,1.20)
else {
    out "\nPlease choose a mode of 1,2,or 3"
}

```

```

</text><value>13.06</value></cell><cell><cname>RELY</cname><text>/*
  * Required Reliaability

```

Rate according to the scale below. Place rating into the value field of this cell.

```

Extra Low      ----
VeryLow        0.75
Low            0.88
Nominal        1.00
High           1.15
VeryHigh       1.40
ExtraHigh      ----
*/

```

```

</text><value>1.150</value></cell><cell><cname>MODP</cname><text>/*
  * Modern Progrmming Practices

```

Rate according to the scale below. Place rating into the value field of this cell.

```

Extra Low    ----
VeryLow      1.24
Low          1.10
Nominal      1.00
High         0.91
VeryHigh     0.82
ExtraHigh    ----
*/

```

```

</text><value>1.100</value></cell><cell><cname>AEXP</cname><text>/*
* Application Experience

```

Rate according to the scale below. Place rating into the value field of this cell.

```

Extra Low    ----
VeryLow      1.29
Low          1.13
Nominal      1.00
High         0.91
VeryHigh     0.82
ExtraHigh    ----
*/

```

```

</text><value>0.9100</value></cell><cell><cname>home</cname><text>/*

```

*The Cocomo model estimates the total effort, schedule, and number of developers required to develop a software product, given the estimated number of thousands of lines of delivered source instructions (KDSI) and the mode of software development.

The estimates provided are:

```

MM -- the total effort required in man months
TDEV -- schedule time reequred assuming optimum staffing
FSP -- fulltime software developers needed for optimum staffing.

```

You must decide whether the project takes place in Organic, Semidetached, or Embedded mode. Visit those three cells for detailed definitions, and set mode's value to 1 for Organic, 2 for Semidetached, or 3 for Embedded.

Enter the number of thousands of estimated delivered source lines into the value field of cell KDSI.

Then return here (home cell) and press the |> button below right.

Go to cell Enhanced for instructions on generating enhanced estimates based upon a n\

umber of detailed factors.

*/

MM TDEV FSP

out "\n MM=\$MM man months, TDEV=\$TDEV calendar months, FSP=\$FSP developers required"

</text><value>1.968</value></cell><cell><cname>CPLX</cname><text>/*

* Product Complexity

Rate according to the scale below. Place rating into the value field of this cell.

Extra Low ----

VeryLow 0.70

Low 0.85

Nominal 1.00

High 1.15

VeryHigh 1.30

ExtraHigh 1.65

*/

</text><value>1.200</value></cell><cell><cname>STOR</cname><text>/*

* Main Storage Constraint

Rate according to the scale below. Place rating into the value field of this cell.

Extra Low ----

VeryLow ----

Low 0.98

Nominal 1.00

High 1.06

VeryHigh 1.21

ExtraHigh 1.56

*/

</text><value>1.000</value></cell><cell><cname>TURN</cname><text>/*

* Turnaround Time

Rate according to the scale below. Place rating into the value field of this cell.

Extra Low ----

VeryLow ----

Low 0.87

Nominal 1.00

High 1.07

VeryHigh 1.15

```

ExtraHigh      ----
*/
</text><value>1.000</value></cell><cell><cname>EnhancedMM</cname><text>if(mode==1)
    3.2*pow(KDSI,1.05)
else if(mode==2)
    3.0*pow(KDSI,1.12)
else if(mode==3)
    2.8*pow(KDSI,1.20)
else {
    out "\nPlease choose a mode of 1,2,or 3"
}
</text><value>93.97</value></cell><cell><cname>TDEV</cname><text>/*
    * This cell computes the number of months estimated for software development, given \
the number of man months of effort required. This assumes the optimum staffing, whi\
ch can be computed from the cell FSP.
*/
if (mode==1)
    2.5*pow(MM,0.38)
else if (mode==2)
    2.5*pow(MM,0.38)
else if (mode==3)
    2.5*pow(MM,0.32)
else {
    out "\nInvalid Mode"
}
</text><value>6.637</value></cell><cell><cname>DATA</cname><text>/*
    * Database Size

```

Rate according to the scale below. Place rating into the value field of this cell.

```

Extra Low      ----
VeryLow        ----
Low            0,94
Nominal        1.00
High           1.08
VeryHigh       1.16
ExtraHigh      ----
*/
</text><value>1.000</value></cell><cell><cname>VIRT</cname><text>/*
    * Virtual Machine Volatility

```

Rate according to the scale below. Place rating into the value field of this cell.

Extra Low	----
VeryLow	----
Low	0.87
Nominal	1.00
High	1.15
VeryHigh	1.30
ExtraHigh	----

*/

```
</text><value>1.000</value></cell><cell><cname>mode</cname><text>/*
```

* The mode of the software development project will be either Organic, Semidetached\, or Embedded. Visit the cells with those names and decide which mode applies to your project.

Then set the value of this cell to: 1 for Organic, 2 for Semidetached, or 3 for Embedded

```
*/</text><value>1.000</value></cell><cell><cname>VEXP</cname><text>/*
```

* Virtual machine Experience

Rate according to the scale below. Place rating into the value field of this cell.

Extra Low	----
VeryLow	1.21
Low	1.10
Nominal	1.00
High	0.90
VeryHigh	----
ExtraHigh	----

*/

```
</text><value>1.000</value></cell><cell><cname>FSP</cname><text>/*
```

* This is the number of full time software developers needed to achieve the development time computed in TDEV.

*/

\$MM/\$TDEV

```
</text><value>1.968</value></cell><cell><cname>Embedded</cname><text>/*
```

* The embedded mode of software development describes projects where there is only \ a general understanding of product objectives, moderate experience in working with related systems, a full need for conformance with pre-established requirements and external interface specs, full concurrent development of hardware and procedures,, considerable need for innovative data processing architectures and algorithms, and a high premium on early completion. This mode can be for all product size ranges.

```
*/</text><value>0.000</value></cell><cell><cname>Organic</cname><text>/*
```

* The organic mode of software development describes projects where there is a thorough understanding of product objectives, there is extensive experience in working with related software systems, there is a basic need for conformance with published r\

requirements and interface specs, some concurrent development of new hardware and operational procedures, minimal need for innovative architectures and algorithms, a low premium on early completion, and less than 50 KDSI.

```
*/</text><value>0.000</value></cell><cell><cname>TIME</cname><text>/*
```

* Execution Time Constraint

Rate according to the scale below. Place rating into the value field of this cell.

Extra Low	----
VeryLow	1.24
Low	1.10
Nominal	1.00
High	0.91
VeryHigh	0.83
ExtraHigh	----

*/

```
</text><value>1.000</value></cell><cell><cname>LEXP</cname><text>/*
```

* Programming Language Experience

Rate according to the scale below. Place rating into the value field of this cell.

Extra Low	----
VeryLow	1.14
Low	1.07
Nominal	1.00
High	0.95
VeryHigh	----
ExtraHigh	----

*/

```
</text><value>1.000</value></cell><cell><cname>TOOL</cname><text>/*
```

* Use of Software Tools

Rate according to the scale below. Place rating into the value field of this cell.

Extra Low	----
VeryLow	1.24
Low	1.10
Nominal	1.00
High	0.91
VeryHigh	0.83
ExtraHigh	----

*/

```
</text><value>1.000</value></cell><cell><cname>Semidetached</cname><text>/*
```

* The semidetached mode of software development describes a project in which there\ is considerable understanding of product objectives, considerable experience in wor\ king with related software systems, considerable need for conformance with pre-estab\ lished requirements and external interface specifications, considerable concurrent d\ evelopment of hardware and procedures, some need for innovative data processing arch\ itectures and algorithms, a medium premium on early completion, and between 50 and 3\ 00 KDSI.

```
*/</text><value>0.000</value></cell><cell><cname>TotEffect</cname><text>$RELY * $DAT\
A * $CPLX * $RQUT * $TIME * $STOR * $VIRT * $TURN * $ACAP * $AEXP * $PCAP * $VEXP * \
$LEXP * $MODP * $TOOL * $SCED</text><value>1.520</value></cell></topic>
```

Cell Script for Function Point Computation

Script written by Kevin J. Northover

Refs: Capers Jones, IEEE Computer, March 1996, 116-118. Ted Lewis, IEEE Computer, August 1996, 13-15.

```
<?xml version="1.0" program="Formulate" encoding="UTF-8"?><topic>
<numfmt>0</numfmt>
<sigdig>4</sigdig>
<fixedec>2</fixedec>
<trigunits>0</trigunits>
<xcell>home</xcell>
<ycell>home</ycell>
<xdivs>1</xdivs>
<ydivs>1</ydivs>
<xmin>0</xmin>
<ymin>0</ymin>
<xmax>1</xmax>
<ymin>1</ymin>
<aplot>1</aplot>
<lwidth>2</lwidth>
<cell><cname>fp</cname><text>/*
* Number of Function Points in the project.
*/
if ($KDSI > 0) {
    $KDSI*1000/$LOC_to_fpoint
}
</text><value>220.00</value></cell><cell><cname>Test</cname><text>/*
* Rule 4: Raising the number of function points to the 1.2 power predicts the appr\
oximate number of test cases created.
*/
```

```

    pow(fp,1.2)
</text><value>647.00</value></cell><cell><cname>Critical_Creep</cname><text>/*
    *This cell computes the monthly creep rate (percent change in requirements) for which the growth in requirements over the build time for the project equals the size of the original specification.
    */
    100*exp(ln(2)/pow($fp, 0.4)) - 1</text><value>107.34</value></cell><cell><cname>Creep</cname><text>/*
    * Rule 3: Creeping user requirements will grow at an average rate of 1 percent per month over the entire development schedule.
    * Enter assumed monthly creep rate in the value field.
    */</text><value>0.01</value></cell><cell><cname>Defect</cname><text>/*
    * Rule 5: Raising the number of function points to the 1.25 (1.27) power predicts the approximate defect potential for new (enhancement) software projects.
    */
    pow(fp,1.25)
</text><value>847.28</value></cell><cell><cname>Words</cname><text>/*
    * Number of delivered words of documentation
    */
    Paper*400
</text><value>197626.91</value></cell><cell><cname>BugsLeft</cname><text>/*
    * Uses Rule 6: Bugfix to estimate the number of defects that will be shipped assuming TestRun test cycles.
    */

    Defect*pow(1-$Bugfix,$TestRun)</text><value>0.00</value></cell><cell><cname>Growth</cname><text>/*
    * Percentage Growth in Requirements by delivery for Creep rate
    */
    100*(pow(1+$Creep,Schedule)-1)
</text><value>8.99</value></cell><cell><cname>Schedule</cname><text>/*
    * Rule 7: Raising the number of function points to the 0.4 power predicts the approximate development schedule in calendar months.
    */
    pow(fp,0.4)</text><value>8.65</value></cell><cell><cname>Bugfix</cname><text>/*
    * Rule 6: Each software review, inspection, or test step will find and remove 30 percent of the bugs that are present.
    */</text><value>0.30</value></cell><cell><cname>home</cname><text>/*
    * Software Estimating Rules of Thumb.

```

This web generates a number of estimates for sizing software projects based on the function point model. These are very rough estimates to be used with CAUTION.

To use, either:

a) enter the number of thousands of lines of delivered source code into KDSI;
or b) set KDSI to 0 and enter the number of function points in fp. Then return to t\
his home cell and execute.

For more refined estimates consider varying the conversion from source lines to func\
tion points LOC_to_fpoint. Or explore the creep rate parameter Creep.

To investigate the number of Quality Control steps required vary the TestRun paramet\
er.

Thanks to: Kevin J. Northover

Refs: Capers Jones, IEEE Computer, March 1996, 116-118.

Ted Lewis, IEEE Computer, August 1996, 13-15.

*/

fp Schedule Paper Words Staffing Growth Test TestRun Critical_Creep Defect
Bugfix BugsLeft Maintenance Lifetime Effort

```
out "1. Function Points      $fp"
out "2. Paper Deliverables  $Paper pages, $Words words"
out "3. Fnct. Pts. Growth   $Growth percent by delivery"
out "3a. Critical Creep     $Critical_Creep percent"
out "4. Test Cases          $Test test cases, $TestRun runs"
out "5. Defect Potential    $Defect"
out "6. Defects Shipped     $BugsLeft"
out "7. Schedule (months)   $Schedule"
out "8. Build Staffing      $Staffing"
out "9. Maintenance Staff   $Maintenance"
out "9b. Years of Use       $Lifetime"
out "10. Total Man Months:   $Effort"
```

</text><value>12.69</value></cell><cell><cname>Effort</cname><text>/*

* Rule 10: Multiply software development schedules by number of personnel to predic\
t the approximate number of months of staff effort.

* Commonly define a staff month as 22 working days with six productive hours each d\
ay, or 132 work hours per month

*/

\$Schedule*\$Staffing</text><value>12.69</value></cell><cell><cname>KDSI</cname><text\
>/*

*

If you estimate the number of function points and put that number into fp, set this \

number to zero.

Otherwise, enter the estimated number of thousands of lines of delivered noncommentary logical source code statements in the value field of this cell.

*/

</text><value>5.50</value></cell><cell><cname>Maintenance</cname><text>/*

* Rule 9: Dividing the number of function points by 500 predicts the approximate number of maintenance personnel required to keep the application updated.

*/

fp/500

</text><value>0.44</value></cell><cell><cname>Lifetime</cname><text>/*

* Rule 9b: Raising the function points total to the 0.25 power yields the approximate number of years the application will stay in use.

*/

pow(fp,0.25)

</text><value>3.85</value></cell><cell><cname>Paper</cname><text>/*

* Rule 2: Raising the number of function points to the 1.15 power predicts approximate page counts for paper documents associated with software projects.

*/

pow(fp,1.15)

</text><value>494.07</value></cell><cell><cname>Staffing</cname><text>/*

* Rule 8: Dividing the number of function points by 150 predicts the approximate number of personnel required for the application.

*/

fp/150

</text><value>1.47</value></cell><cell><cname>TestRun</cname><text>/*

* Number of Test Cycles in development of system. Includes Major Design Reviews, code inspections and various testing levels.

*/

Test*4.0

</text><value>2588.02</value></cell><cell><cname>LOC_to_fpoint</cname><text>/*

* Rule 1: One function point = 100 logical source code statements.

This ratio can vary from >300 for assembler to <60;20 for object oriented languages and program generators. For procedural languages such as COBOL, C, Fortran, etc. 100 is a rough conversion factor. Just a guess that conversion factor is 25 for embedded systems.

Enter the conversion factor to use in the value field.

*/

</text><value>25.00</value></cell></topic>

APPENDIX B

Vendor Documents

The documents listed below show where to buy components for the digital trombone or obtain data about components.

STM32 Nucleo-32 development board with STM32L432KC MCUProduct Page:

<https://www.st.com/en/evaluation-tools/nucleo-l432kc.html>

STM32 Nucleo-32 User Manual (UM1956):

https://www.st.com/content/ccc/resource/technical/document/user_manual/e3/0e/88/05/e8/74/43/a0/DM00231744.pdf/files/DM00231744.pdf/jcr:content/translations/en.DM00231744.pdf

STM32CubeL4 Vendor Software Support:

https://www.st.com/content/st_com/en/products/embedded-software/mcus-embedded-software/stm32-embedded-software/stm32cube-mcu-packages/stm32cubel4.html

NUCLEO-L432KC DigiKey Product Page:

<https://www.digikey.com/product-detail/en/stmicroelectronics/NUCLEO-L432KC/497-16592-ND/6132763>
Price 10.99 € (half the Amazon price).

Honeywell product page for pressure sensor:

<https://sensing.honeywell.com/ABPDANN005PG2A3-amplified-board-mount-pressure-sensors>

Pressure Sensor i2c specification data sheets:

<https://sensing.honeywell.com/i2c-comms-digital-output-pressure-sensors-tn-008201-3-en-final-30may12.pdf>

https://sensing.honeywell.com/index.php?ci_id=45841

Sparkfun 3.5mm audio jack breakout:

<https://www.sparkfun.com/products/11570>

Slide Potentiometer Module:

<https://robu.in/product/sliding-adjustable-potentiometer-module-10k/>

Bibliography

- 1 The Unified Modeling Language User Guide, Booch, Rumbaugh, Jacobson, 1999, Addison-Wesley.
- 2 Object Oriented Software Engineering, Ivar Jacobson, 1992, ACM Press, Addison-Wesley.
- 3 Managing Software Requirements: A Unified Approach, Leffingwell, Widrig, 1999, Addison-Wesley.
- 4 Software Cost Estimation with COCOMO II, Barry Boehm, 2000, Prentice-Hall.
- 5 Function Point Analysis, Garmus, Herron, 2000, Addison-Wesley.
- 6 Object Oriented Design, Grady Booch, 1991, The Benjamin/Cummings Publishing Company, Inc.
- 7 Software Testing Techniques, Beizer, 1990, Van Nostrand Reinhold.